TURING 图灵电子与电气工程丛书

ELSEVIER

Microchip公司
全套课程解决方案首选图书

PIC嵌入式系统开发

Designing Embedded Systems with PIC Microcontrollers

Principles and Applications

[英] Tim Wilmshurst 著
陈小文 闫志强 等译







人民邮电出版社
POSTS & TELECOM PRESS

TURING 图灵电子与电气工程丛书

PIC嵌入式系统开发

Designing Embedded Systems with PIC Microcontrollers
Principles and Applications

[英] Tim Wilmshurst 著
陈小文 闫志强 等译

Designing Embedded Systems with PIC Microcontrollers
Principles and Applications

Tim Wilmshurst



人 民 邮 电 出 版 社
北 京

图书在版编目(CIP)数据

PIC嵌入式系统开发/(英)威尔姆舍斯特(Wilmshurst,T.)著;陈小文等译. —北京:人民邮电出版社,2008.9
(图灵电子与电气工程丛书)
书名原文:Designing Embedded Systems with PIC Microcontrollers:Principles and Applications
ISBN 978-7-115-18265-4

Ⅰ.P… Ⅱ.①威… ②陈… Ⅲ.单片微型计算机,PIC系列-系统开发 Ⅳ.TP368.1

中国版本图书馆CIP数据核字(2008)第082992号

内容提要

本书系统而全面地介绍了嵌入式系统设计的原理及其应用,包括嵌入式系统的指令集系统结构、流水线、存储设备、定时器、中断、时钟、并行串行通信、互连网络、开发环境和开发语言等重要内容。书中对嵌入式系统设计的讲解主要以Microchip公司的3款PIC微控制器(16F84A、16F873A和18F242)为基础,并辅以大量的设计实例。全书编排合理,叙述由浅入深,生动活泼。

本书适合嵌入式系统开发工程师阅读,也可作为高等院校电子、机电和计算机工程相关专业嵌入式系统课程的教材或参考书。

图灵电子与电气工程丛书
PIC嵌入式系统开发

◆ 著 [英] Tim Wilmshurst
译 陈小文 闫志强 等
责任编辑 朱 巍

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子函件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京铭成印刷有限公司印刷

◆ 开本:700×1000 1/16
印张:34.5
字数:716千字 2008年9月第1版
印数:1-4 000册 2008年9月北京第1次印刷

著作权合同登记号 图字:01-2007-5569号
ISBN 978-7-115-18265-4/TN

定价:69.00元

读者服务热线:(010)88593802 印装质量热线:(010)67129223
反盗版热线:(010)67171154

译 者 序

本书作者 Tim Wilmshurst 是英国德比大学电子与声学系的教授，同时也是嵌入式系统方面的专家。他的主要研究领域是电子技术和嵌入式系统，同时，他在 PIC 微控制器的应用上也有很深的造诣。为了辅助教学，他带领学生使用 PIC 微控制器设计了一个自动装置——Derbot AGV，通过该自动装置帮助学生学习嵌入式系统，并获得了巨大的成功。本书是 Wilmshurst 教授使用 Microchip 公司的 PIC 微控制器进行嵌入式系统设计的经验总结

本书介绍了嵌入式系统原理和应用。书中结合 PIC 系列的 3 个微控制器(16F84A、16F873A 和 18F242)分阶段地讨论了 PIC 微控制器的设计思想，进而阐明通过 PIC 微控制器设计嵌入式系统这一主题。本书结构合理，可读性强，循序渐进地讲解了嵌入式系统设计的基本原理及其应用，书中还有大量的图、表和示例，这形成了本书的一大特色。通过对本书的学习，读者可在较短时间内轻松地掌握当今嵌入式系统软硬件的基本知识和技能。无论是对于工程技术人员、学生还是嵌入式系统爱好者来说，本书都是一个不错的选择。Microchip 公司的 PIC 微控制器广泛地应用于嵌入式系统领域，希望本书能为使用 PIC 微控制器的嵌入式系统开发人员提供帮助。

本书主要由陈小文和闫志强翻译。此外，参与翻译的人员还有：肖枫涛、刘齐军、林龙信、李晋文、张聪、韩智文、马蓉、焦贤龙、邝祝芳、奚丹、刘志忠、陈钢、宋锐、石志广、唐玲艳、唐扬斌、叶俊、杨明军、张杰良、颜炯、薄建禄、肖国尊等。

由于译者水平有限，加之时间紧迫，文中难免有翻译不当或欠妥之处，敬请读者批评指正。

译 者

于国防科技大学计算机学院

2008 年 6 月

前言

本书是一本嵌入式系统方面的图书，主要通过3个PIC微控制器的应用实例，从初级到高级、循序渐进地介绍了嵌入式系统。在初级部分，本书力图使读者能够通过阅读本书从而胜任嵌入式系统领域的工作，成为一名独立的设计人员；在高级部分，读者将获得在嵌入式领域进行专业实践的必要技能。

为了达到上述目标，本书介绍了当前嵌入式系统软硬件开发的基本知识和技能。在硬件方面，深入研究了微控制器设计以及与微控制器接口相关的电路和变换器。在软件方面，则同时介绍了汇编语言编程和C语言编程。本书最后还介绍了实时操作系统的研究和应用。

本书由5个部分构成。中间3个部分为主体，主要围绕3个PIC微控制器——16F84A、16F873A和18F242展开讨论。全书对这3个实例逐个进行讲解，并由此引出本书所要介绍的一系列复杂的思想。然而，需要指出的是，不应当将本书看作一本关于PIC微控制器的技术手册，这是一本探讨嵌入式设计的图书。读者可以从这3个PIC的应用实例中学到必要的技能和知识，并能举一反三，融会贯通。

本书的一大特色是实践和理论相结合。不论是在硬件上还是在程序仿真上，书中大部分内容都辅以实际应用进行描述，而并不只是仅有抽象的理论知识。本书首先介绍了一个非常简单的电子乒乓球游戏项目。随后介绍了全书的主要项目Derbot AGV(Autonomous Guided Vehicle，自主导向车)，这是一个用户定制系统，可作为一款独立的开发平台使用。AGV可以开发成不同的形式，也可以应用于波形发生器、电子测距仪、测光仪等诸如此类的众多仪器仪表中。读者可以利用本书附属资源[①]中所提供的设计信息来建立书中所有项目实例。另一方面，这些项目也可以作为理论实例来研究。

本书主要供高等院校电子、机电和计算机等相关专业二三年级本科生使用，也适用于相关领域的爱好者和专业人士阅读。读者在阅读本书之前需要有一定的电子专业知识，也就是说，学过本科一年级课程，了解晶体管和二极管的运行机制，懂得简单的模拟和数字电子子系统。对计算机体系结构也要有一定了解，例如学过有关微处理器方面的初级课程，这对使用本书也是很有帮助的。

本书分为初级、中级和高级3个部分，因此，它可作为多门课程或教学单元的教

---

① 本书附属资源请登录图灵网站(www.turingbook.com)本书配套网页免费下载。——编者注

材。前 6 章为初级部分，介绍了微控制器及其汇编编程，可以作为一个学期的学习课程。这几章是以 16F84A 作为实例进行讲述的。16F84A 简单易学，是一款优秀的初级微控制器。第 7 章到第 11 章构成了本书的中级部分，介绍了使用汇编语言对更为复杂的系统进行编程，以 16F873A 作为实例，详细讲述了微控制器外围设备的详细知识及其应用。第 12 章到第 20 章作为高级部分，以 18F242 为实例，讲述了 C 语言编程以及 RTOS 的使用。任课教师可以根据他们对 C 语言、汇编语言以及微控制器使用的偏好有选择地讲授第 7 章到第 20 章。也可以继第 1 章到第 6 章之后，略过第 7 章到第 11 章，直接教授第 12 章。不介绍中间这几章的详细内容不会影响授课。因为中级部分和高级部分所采用的编程方法大致相同，只是与汇编相比，使用 C 语言可以不必过多关注外围设备的详细信息。高级部分中，会在需要的地方引用中级部分的知识，读者可以据此返回查找相关内容。

无论读者选择何种阅读顺序，都至少应该做好了使用本书附属资源所提供的 Microchip MPLAB® 集成开发环境的准备。读者可以通过这个集成开发环境模拟书中的实例程序，然后在此基础上进行修改和进一步开发。本书不可避免地要从介绍一些硬件知识开始，以便读者了解运行软件的硬件系统的基本知识。在某种程度上，前面几章关于 PIC 微控制器体系结构的介绍对初学者来说包含了丰富的知识，需要花大量时间去学习。有这几章基础知识的铺垫，读者在第 4 章就会开始享受到编程和仿真带来的乐趣，同时还可以学习到软硬件知识并了解软硬件之间如何协同工作。本书的最后一部分将用到 Microchip C18 的 C 编译器，本书附属资源提供了这款编译器的学生版本和第 19 章使用的 Salvo™ RTOS 的 Lite 版本。

除了进行程序仿真外，读者无论是在家里、学校还是在工作场所，都应该尽可能地观察电子元件和测试设备。使用这些电子元件和测试设备，他们可以在同样的系统上建立或运行一些项目实例，只有这样做才能完成实时嵌入式系统的实际实现。当课程进行到中间和后面部分的章节时，任课老师最好给每个学生分发一块 Derbot 印制电路板和一组基本的电子元件，指导学生进行最初的系统开发，然后给学生提出更进一步的定制建议。学生们能随后提出一些想法就更好了。本书附属资源给出了定制系统的设计细节。

依据制造厂提供的数据手册进行相应的嵌入式开发是嵌入式领域专业设计人员的一项必备技能。这些数据手册是采用微控制器进行设计的主要信息来源，也是专业领域的最好参考。通常，从第三方获得的图表往往不尽如人意，即使这些图表是数据手册中的信息的简化版。所以，本书经过授权直接采用了 Microchip 公司数据手册中的大量图表，且多数图表加入了注解以便读者阅读。我们鼓励读者从网上下载数据手册的完整版本来查阅并使用。

嵌入式系统由很多方面组成，包括硬件、软件、半导体技术、模拟和数字电子学、计算机体系结构、传感器和执行器等，因此，要想具备完整的嵌入式系统知识，需要对这些知识进行广泛而深入的学习。本书重点介绍了 PIC 微控制器，不可能涉及嵌入式领

域的方方面面。

希望你能够愉快地阅读完这本书，更希望你能够享受设计和构建嵌入式系统所带来的挑战和乐趣！

Tim Wilmshurst
英国，德比大学

致谢

书中某些材料已获得版权持有者 Microchip 技术公司的授权。版权所有，违者必究。未经 Microchip 技术公司书面同意禁止重印或复制。

PIC®、PICSTART®和 MPLAB®都是 Microchip 技术公司的注册商标。PICBASIC™、PICBASIC PRO™、ECAN™、In-Circuit Serial Programming™、ICSP™、MPASM™、MPLIB™、MPLINK™、MPSIM™和 PICDEM. net™都是 Microchip 技术公司的商标。

图 1-11、图 1-13、图 2-2～图 2-10、图 3-8、图 3-10～图 3-12、图 3-14～图 3-16、图4-4、图 4-13、图 6-2、图 6-3、图 6-8～图 6-10、图 7-1～图 7-4、图 7-6、图 7-7、图 7-9～图 7-11、图 7-14～图 7-16、图 7-25、图 7-26、图 8-7、图 9-1、图 9-2、图 9-4、图 9-5、图 9-7～图 9-9、图 9-11、图 10-7～图 10-9、图 10-14～图 10-21、图 10-25～图 10-28、图 11-6～图 11-10、图 11-15、图 12-1～图 12-10、图 12-13、图 12-14 和图 13-1～图 13-11 都来自于 Microchip 公司的以下数据手册：PIC12F508/509/16F505(DS41236A)、PIC16F84A(DS35007B)、PIC16F87XA(DS39582B)、PIC18FXX2(DS39564B)和 PIC micro™ Mid-Range MCU Family Reference Manual(DS31004A 和 DS31005A)。所有这些图片都获得了 Microchip 技术公司的授权。

非常感谢 David Manley 和 Mike Vernon 给我公休假，这才使我能够完成此书。感谢包括 Jonathan Guinet、David Coterill-Drew、Grigorios Dedes 和 Kelvin Brammer 在内的众多学生，他们设计了自己的 AGV 并且为 Derbot 项目提出了不少意见。也感谢 Naoko Evans 和 Nick Roberts，他们阅读了本书手稿的各个章节并提出了很多中肯的意见。感谢 Trevor Noble，多年来他运用自己熟练的技能，充满热情地设计了包括一系列 Derbot 原型在内的众多嵌入式系统。感谢 Microchip 技术公司的全体员工，他们为我解答了许多技术上的问题，并解决了关于版权和其他方面的问题，并且慷慨地授予了许多版权材料的复制权。同样感谢 Salvo™ Operating System 的开发商 Pumpkin 公司，感谢他们给我提供了技术支持并允许我们将 Salvo“Lite”放入本书的附属资源中。Salvo™是 Pumpkin 公司的商标。

特别感谢我的家人 Beate、Imogen、Jez 和 Naomi，感谢他们这 15 个月来对我的支持，这 15 个月已成为我们生活中一个非常重要的部分。和他们在一起，我的生活充满了快乐，在此，将此书献给我的家人。

目　录

第一部分　嵌入式系统入门

第二部分　最小的系统和 PIC® 16F84A

第五部分 网络互连技术

第一部分　嵌入式系统入门

本部分为绪论，介绍了嵌入式系统和微控制器，并由此纵览了 Microchip 公司旗下的 PIC® 微控制器。

第 1 章 微小的计算机，隐藏的控制

我们生活在信息革命的时代，使用着功能异常强大的计算机。目前，计算机已经深入到了我们生活中的各个领域。一些计算机为了满足工业和研究中的高性能应用而不计成本地被设计成尽可能地强大；另一些计算机用于家庭和办公，处理能力欠佳但价格较低。还有一类计算机不为人们所知，之所以会这样，某种程度上是因为看不见它。这类计算机被设计安放在产品里，用于控制该产品。这类计算机是隐藏不可见的，以至于用户经常不知道它们就在产品里。包含此类计算机的产品称为嵌入式系统，这也正是本书将要讲授的内容。我们通常称那些微小的计算机为微控制器，本书所要讲述的产品就属于微控制器。

本章将介绍以下内容：

- "嵌入式系统"一词的含义；
- 位于嵌入式系统核心部位的微控制器；
- Microchip PIC® 系列；
- 第一款 PIC 微控制器——12F508；
- 来自 Freescale 公司的另一种微控制器结构。

1.1 当今嵌入式系统概述

什么是嵌入式系统

嵌入式系统的基本思想很简单。如果我们有项目产品需要控制，而该产品中内嵌了一个计算机来提供这样的控制，那么我们就有一个嵌入式系统了。嵌入式系统可以定义为[1.1]：

一个系统，它的主要功能不是计算，而是接受内嵌于其中的计算机的控制。

现今嵌入式系统无所不在，广泛出现在家庭、办公室、工厂、汽车和医院中。表 1-1 列举了一些可能是嵌入式系统的产品例子，所有这些例子都广为人知。虽然大多数例子之间差异非常大，但就它们作为嵌入式系统的特性而言，均有相同的设计原理。

绝大多数用户不会认识到他们所使用的产品由一个或多个嵌入式计算机所控制。事实上，虽然都见过控制计算机，但他们几乎不会认为那是计算机。毕竟，大多数人是通过屏幕、键盘、磁盘驱动器等认识计算机的，而嵌入式计算机并没有这些设备。

表 1-1 一些熟知的嵌入式系统例子

家庭	办公室和商业场所	汽车
洗衣机	复印机	车门机械装置
电冰箱	结账机	暖通空调
防盗自动警铃	打印机	刹车控制
微波炉	扫描仪	引擎控制
中央暖气控制器		车内娱乐
玩具和游戏		

1.2 一些嵌入式系统例子

让我们看一些嵌入式系统的例子，前面一些来自于日常生活，后面一些则来自于本书所讨论的项目。

1.2.1 家用电冰箱

如图 1-1 所示是一个简单的家用电冰箱，其内部需要维持一个适度稳定的低温。通过感知电冰箱内部温度并与所需温度相比较，可以达到维持稳定低温的目的。通过接通压缩机可以降低电冰箱内部的温度。测量温度需要一个或多个传感器，还需要信号调整和数据获取电路。通过某种数据处理来比较代表测量温度的信号和代表所需温度的信号，然后产生控制输出。控制压缩机需要某种形式的电子接口，这些电子接口接受低电平输入控制信号，再将这些输入信号转化为必要的电气驱动，从而开关压缩机的电源。

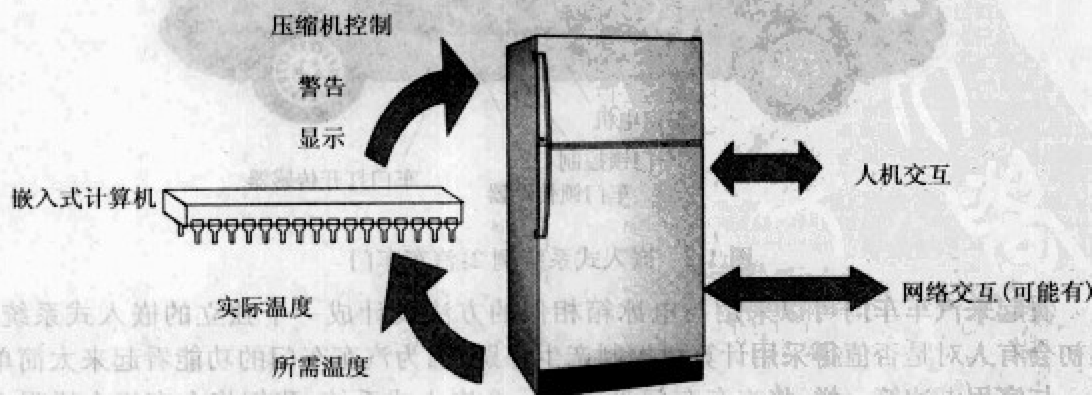


图 1-1 嵌入式系统例 1：电冰箱

4

电冰箱的控制过程可以通过一个传统的电子电路来完成，亦可以通过一个小型嵌入式计算机来完成。如果使用嵌入式计算机来完成，那么仅需简单的设计就可实现上

述最低限度要求的控制过程。然而，一旦在电冰箱适当的位置加入小型计算机，就会有极大的机会来提高电冰箱的附加值。由于采用了数字信号且目前处理能力很强，因此，在电冰箱上加入一些诸如智能显示、更高级的控制特征、更好的用户控制机制之类的特征非常容易。

下面进一步讨论关于通过加入小型计算机来提高电冰箱附加值的问题，一旦在电冰箱适当的位置加入了嵌入式计算机，那么将它与其他嵌入式计算机或者其他形式的计算机通过网络互连在一起是可能的。这开启了巨大的新的应用范围，允许一个小的系统作为一个较大系统的子集，并与这个大的系统共享信息。现在这种情况正发生在像电冰箱这样的家用产品以及其他更复杂的产品上。

图 1-1 虽然仅描述了电冰箱，但实际上它很好地阐述了嵌入式系统的全面概念。电冰箱中，有一个嵌入式计算机用于读入内部变量和输出信号以控制系统的性能。可能有人机交互（通常这项不是必要的）和网络交互。通常情况下，用户并不知道在电冰箱内部存在计算机。

1.2.2 汽车车门机械装置

一个非常复杂的嵌入式系统例子是汽车车门，如图 1-2 所示。它同样有若干传感器和人机交互装置，以及一组必须响应系统需求的传动装置。一组传感器感应车门锁信息，而另一组传感器感应车窗信息。有两个传动装置，分别为车窗电机和锁传动器。



图 1-2 嵌入式系统例 2：汽车车门

5

看起来汽车车门可以采用与电冰箱相似的方法设计成一个独立的嵌入式系统。起初会有人对是否值得采用计算机控制产生怀疑，因为汽车车门的功能看起来太简单了。与家用电冰箱一样，将汽车车门设计为一个嵌入式系统，我们将会有机会增强它的功能。现在，已经有了在电子控制下的车门状态和车门制动器，它们可以和汽车的其余部分集成在一起。如果汽车司机设法关门而车门无法锁上，就可以通过引入中央锁控锁上车门或者通过报警装置发出警告。因此，采用网络将车门控制的简单操作与

汽车的其他功能连在一起，这会带来巨大的好处。在后续各章我们将看到，网络交互是嵌入式系统的一个重要特征。

1.2.3 电子乒乓球

图1-3描述了一个小游戏，它是本书几个项目实例之一。这个游戏需要两位选手参加，每位选手有一个称为"球拍"的按钮。选手可通过按下按钮开始游戏。乒乓球用一行共8个发光二极管(Light-Emitting Diode，LED)表示，游戏开始时，球会迅速地朝对方选手飞去，对方选手必须在球恰好位于最后一个发光二极管时按下自己的按钮，而不能在其位于其他位置时按下。如果操作正确，球就会返回，游戏继续，直到有一方犯规为止。一旦有一方犯规，与未犯规选手的分数相关的发光二极管就会被点亮。当球出界时，"出界"发光二极管会被点亮。

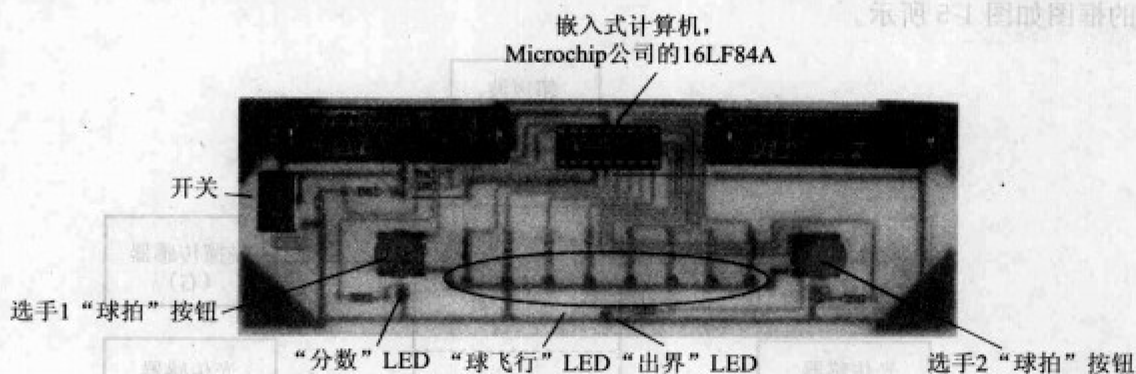


图1-3 电子乒乓球

上述所有的动作通过Microchip公司一个微小的嵌入式计算机——微控制器来控制[1,2]。这个微控制器采用18引脚的集成电路(Integrated Circuit，IC)的形式，它没有计算机通常所具有的外部特征。然而，当今电子技术非常先进，在这样一个小小的集成电路中也存在一个CPU(Central Processing Unit，中央处理器)、一个复杂的存储器阵列以及一组定时电路和接口电路。其中有一个存储器用来存储程序，微控制器通过执行该程序来运行游戏。微控制器能够读入开关(选手的"球拍"按钮)输入的位置，并计算出对应发光二极管的位置，然后驱动与它相连的发光二极管。所有这些计算动作仅仅通过2个AAA电池提供动力。

6

1.2.4 Derbot 自主导向车

本书后续章节介绍的另一个项目是Derbot自主导向车(Autonomous Guided Vehicle，AGV)，如图1-4所示。与我们迄今所见的嵌入式系统例子相比，它的特征是怎样的呢？在照片中可以看到，在它的前面布满了传感器和传动装置。如果Derbot撞到了障碍物，两个微型开关碰撞探测器就会受到感应。安装在伺服传动装置上的超声波探测器可以确保Derbot不会发生不希望的碰撞。伺服传动装置每边各有一个光传感器，

它的后面还有第三个光传感器(在照片中看不见),这些光传感器用来跟踪光线。它还有一个提供导航功能的罗盘,通过这个罗盘可以进行地球磁场定位。两个直流齿轮电机可以提供动力来供其移动,每个电机上有一个传感器(在照片中看不见)记录轮子的旋转次数,以此来计算实际移动的距离。我们可通过以不同的速度驱动轮子旋转来操纵 Derbot。AGV 中含有一个压电发声器,用于提醒使用人员。Derbot 通过 6 个 AA 碱性电池提供动力,这些电池放在轮子上面的电源组里。Derbot AGV 的框图如图 1-5 所示。

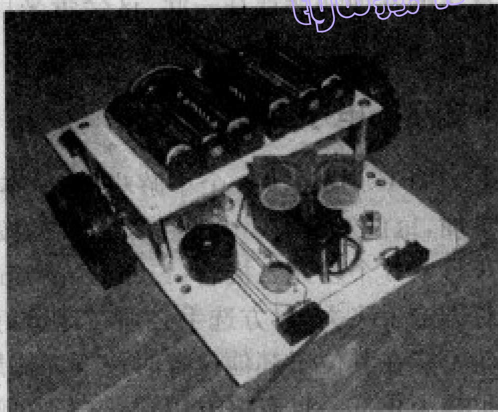


图 1-4 Derbot AGV

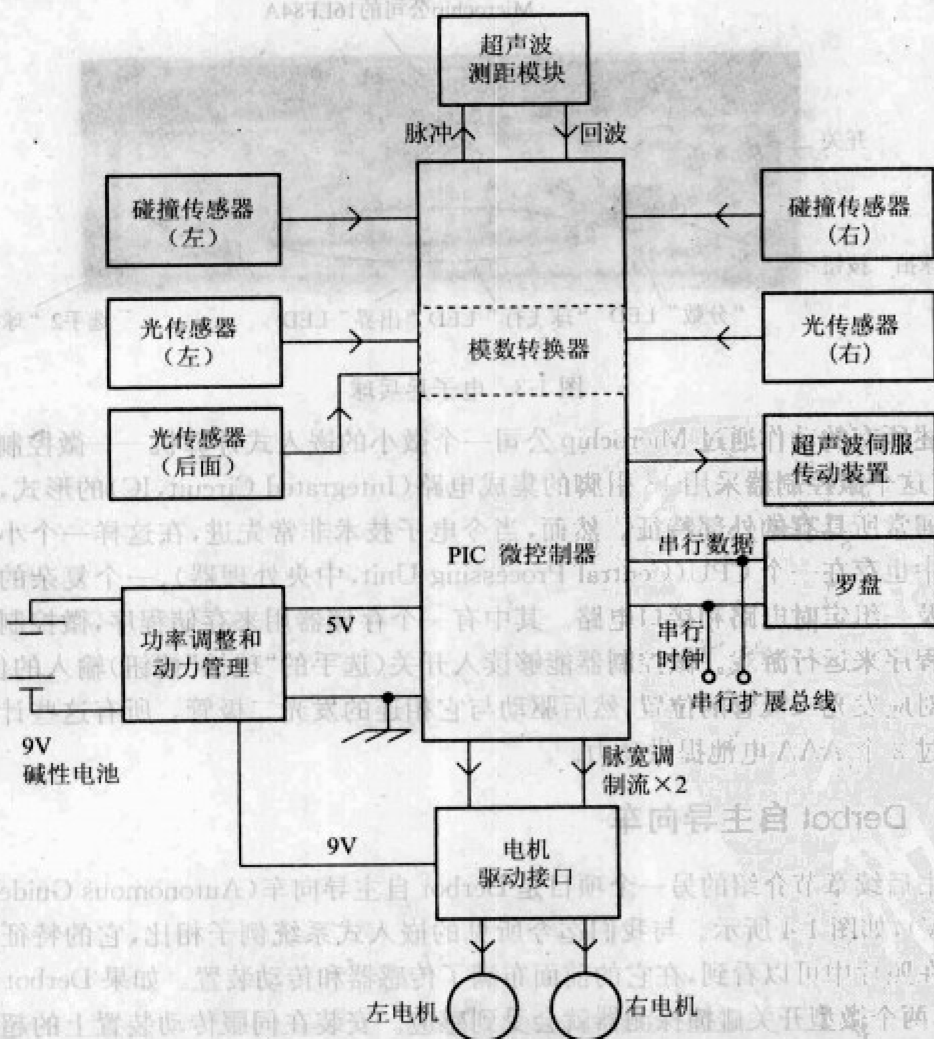


图 1-5 Derbot 框图

与之前介绍的例子相同，Derbot 作为一个嵌入式系统运转，从它的各种传感器中读入数据，然后计算输出到它的传动装置。它由 Microchip 公司的另一款微控制器控制，这款微控制器由于被电源组挡住，所以从照片中看不见。该款微控制器似乎比乒乓球游戏中的微控制器功能更强大，因为它需要连接更多的输入，以更复杂的方式去驱动它的输出。

7

下面我们将会发现一个有趣的现象，每个微控制器的 CPU 都是一样的。它们的主要差别在于它们的接口能力不同。正是这种不同使得 Derbot 微控制器拥有更强的功能。

1.3 一些必备的计算机知识

当我们设计嵌入式系统时，通常需要了解所使用的嵌入式计算机的某些详细特征。这与使用台式计算机有很大的不同，台式计算机用于文字处理或计算机辅助设计，它的内部工作原理巧妙地被隐藏了。为了扩展知识面，让我们先快速浏览一些重要的计算机特征。

8

1.3.1 计算机的组成元素

图 1-6 显示的是计算机系统必备的组成元素。从根本上讲，计算机必须能够进行算术和逻辑运算。这些功能由 CPU 来提供。CPU 通过存储在存储器中被称为程序的一系列指令来工作。虽然这些指令中的任何一条都只执行一个非常简单的功能，但是由于典型计算机运行速度非常快，一旦指令全部执行就会完成非常强大的计算能力。许多指令都会产生数学或逻辑操作，这些操作发生在 CPU 中的算术逻辑单元（Arithmetic Logic Unit，ALU）。

为了起到作用，计算机必须能够同外界通信，它通过其输入/输出来做到这一点。对于个人计算机，这就意味着通过键盘、视频显示装置（Visual Display Unit，VDU）和打印机进行人机交互。在嵌入式系统中，更主要地是通过传感器以及传动装置与它周围的物理世界进行通信。

正在发生的计算机革命不仅应归功于我们现在能做到的惊人的处理能力，还应归功于同样惊人的数据存储和访问能力。一般来说，计算机中的存储器有两种主要的应用，如图 1-6 所示。一种存储器保存计算机将要执行的程序。这种存储器需要永久保存程序，这样，无论上电与否，程序都能保存，并且准备好一旦上电就能马上运行。另一种存储器用于存储在程序运行时使用的临时数据。这种存储器不需要永久保存数据，即使永久保存数据对这种存储器没有任何害处。

最后，这些主要模块之间必须存在数据通路，如图 1-6 中宽箭头所示。

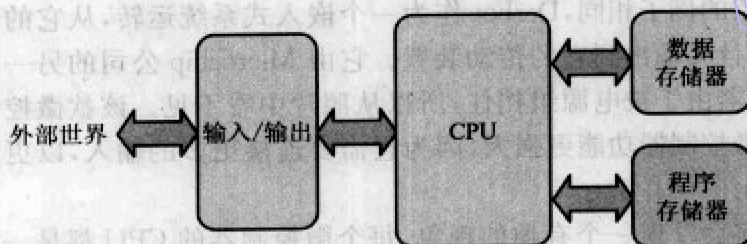


图 1-6　计算机的必备组成元素

1.3.2　指令集——CISC 和 RISC

任何 CPU 都有它自己识别和响应的指令集，所有程序都必须按照该 CPU 的指令集来建立。我们希望计算机执行代码的速度越快越好，但是要想达到这个目的却并不简单。一种方法是构造复杂 CPU，CPU 的指令集包含各种各样的指令，对每一个可预知的操作都有一条指令去处理。这就产生了**复杂指令集计算机**(the Complex Instruction Set Computer, CISC)。CISC 的指令很多而且非常复杂，这种设计的复杂性需要必然会导致运行速度的降低。CISC 方法的一个特征是指令有复杂的层次。简单的指令用短的指令代码表示，如 1B 的数据，而且这些指令的执行速度快。复杂的指令可能需要多个字节的数据去定义它们，并且执行时间很长。

另一种方法是构造非常简单的 CPU，CPU 只需有限的指令集。这就产生了**精简指令集计算机**(the Reduced Instruction Set Computer, RISC)。因此，对于整个设计，指令集非常简单，这会使得 CPU 的运行速度很快。RISC 方法的一个特征是每条指令包含在一个二进制字中。这个二进制字必须保存所有必需的信息，包括指令代码本身以及所需要的地址和数据信息。RISC 方法的另一个特征，也是这种精简指令方法所导致的一个结果，就是通常每条指令的执行时间都相同。

1.3.3　存储器类型

从传统上讲，存储技术分为两类：

(1) **易失性**。这种类型的存储器只有在上电时才能工作，一旦掉电，它将会丢失所有存储的数据，所以它只能用于存储临时数据。一般来说，这种类型的存储器采用简单的半导体技术，从电气特性上讲，其数据易于写入。由于历史原因，这种类型的存储器通常称为随机访问存储器(Random Access Memory, RAM)。它还有一个更为贴切的名字是“数据存储器”。

(2) **非易失性**。这种类型的存储器在掉电的时候仍能保存它所存储的数据。台式计算机的硬盘主要采用这种非易失性的功能来保存大量的数据。嵌入式系统中也采用这种非易失性半导体存储器。制作这种非易失性存储器比较困难，需要用到复杂的半导体技术。通常，从电气特性上看，这种类型的存储器在写入数据时较为困难，要考虑到例如时间开销、功耗、写入过程的复杂性等因素。非易失性存储器用于保存计算

机程序，由于历史原因，通常称为只读存储器（Read-Only Memory，ROM），一个更贴切的名字是“程序存储器”。

随着存储技术越来越成熟，这两种存储器在功能上的差异也越来越小。在第 2 章我们将再次讨论存储技术及其应用。

1.3.4 存储器组织结构

与存储器交互时，需要知道两种类型的信息，一是存储单元的地址，二是存放在存储单元中的实际数据。这两种信息的连线遍布于存储器中，连接每个存储单元，并在存储器外部与两组分别称为地址总线和数据总线的互连线相连。必须确保通过数据总线和地址总线（或者是一部分地址线）能够访问到每个存储单元的数据。

满足上述存储器组织结构要求的存储器的一个简单实现方式如图 1-7a 所示。这个结构由冯·诺依曼发明，因此称为冯·诺依曼结构。这个计算机只有一条地址总线和一条数据总线，要访问程序存储器和数据存储器都需要通过相同的地址总线和数据总线。输入/输出设备也采用这种方式连接，从 CPU 的角度来看，它就像存储器一样。

10

图 1-7b 为与冯·诺依曼结构相对的另一种存储器组织结构。每个存储器区域有自己的地址总线和数据总线。由于这种结构是在哈佛大学发明的，因此称之为哈佛结构。

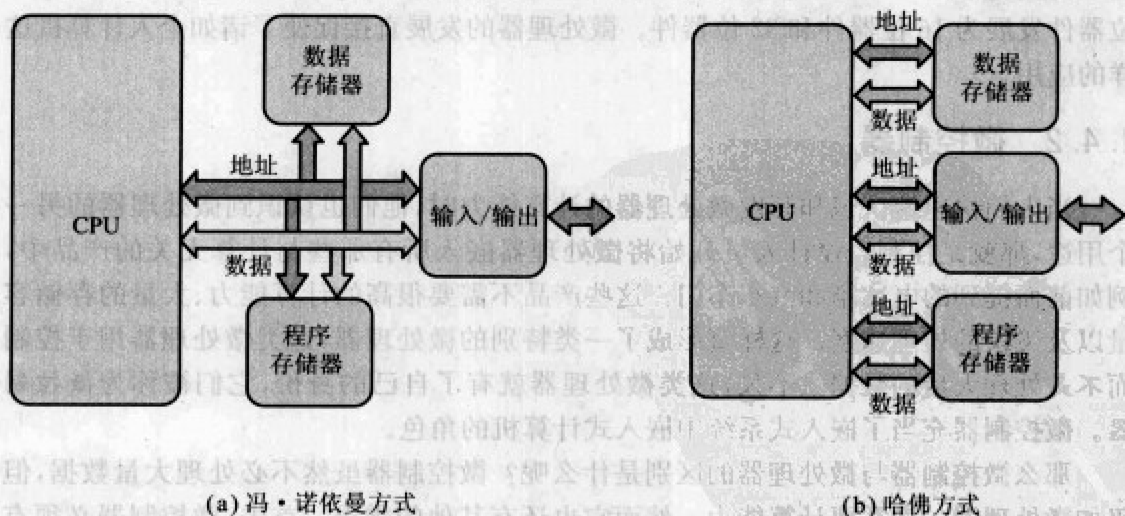


图 1-7 存储器访问组织结构

冯·诺依曼结构简单且符合逻辑，又有某种灵活性。在程序存储器和数据存储器之间可以以任何方式划分可寻址存储器区域。但是，冯·诺依曼结构有两个缺点。一是它是一种“所有区域同一大小”的方法，对所有存储器区域使用同一数据总线，即使某个存储器需要处理长字而另一个存储器需要处理短字。二是由于它共享所有的程

序和数据，因此如果一个人在使用存储器，那么另一个人就不能再使用它了。因此，如果 CPU 正在访问程序存储器，那么数据存储器就必须空闲，反之亦然。

在哈佛结构中，我们在总线宽度上有更大的灵活性，但同时也增加了其复杂性。由于程序存储器和数据存储器各有自己的地址和数据总线，所以每条总线根据需要有不同的宽度，并且程序存储器和数据存储器可以同时被访问。从反面来看，哈佛结构加大了程序存储器和数据存储器之间的差异，即使有些时候这种差异是不希望存在的。哈佛结构也有它的缺点，例如，数据是作为表存放在程序存储器中的，但是它实际上应该属于数据区域。

1.4 微处理器和微控制器

1.4.1 微处理器

最早的微处理器出现在 20 世纪 70 年代。它们是惊世之作，第一次实现了在单一的 IC 上放置一个计算机 CPU，第一次在相当小的空间中以非常低的成本获得了巨大的处理能力。起初，诸如存储器和输入/输出接口之类的所有其他功能都在微处理器的外围，所以一个能工作的系统仍然不得不由大量的 IC 组成。逐渐地，微处理器变得更加独立，具有包含其他功能的可能性，例如像包含 CPU 一样在同一块芯片上包含不同类型的存储器。同时，CPU 的处理能力变得更强大并且速度更快，它也迅速地从 8 位器件发展为 16 位器件和 32 位器件。微处理器的发展直接促使了诸如个人计算机这样的应用。

11

1.4.2 微控制器

当人们快速地认识和开发微处理器的计算能力时，他们也认识到微处理器的另一个用途，那就是控制。设计人员开始将微处理器嵌入所有那些与计算无关的产品中，例如前面提到的电冰箱和汽车车门。这些产品不需要很高的计算能力、大量的存储容量以及飞快的处理速度。这样就形成了一类特别的微处理器，这类微处理器用于控制而不是处理大量的数据。不久，这类微处理器就有了自己的身份，它们被称为微控制器。微控制器充当了嵌入式系统中嵌入式计算机的角色。

那么微控制器与微处理器的区别是什么呢？微控制器虽然不必处理大量数据，但仍如微处理器一样需要计算能力。然而它也还有其他的需求。首先，微控制器必须有卓越的输入/输出能力，这样才能与外部的输入/输出相连接，例如电冰箱或汽车车门中的微控制器与电冰箱或汽车车门的输入和输出相连接。由于许多嵌入式系统特别注意其大小和制作成本，因此微控制器必须是小的、独立的和低成本的，它也不会处于传统计算机所要求的良好控制环境中。微控制器必须承受工业或机动车环境的严酷条件，并且能够在极端温度下工作。

12

图 1-8 是微控制器的总图。从本质上看，它包含一个简单的微处理器核，以及所有

必要的数据存储器和程序存储器。为了形成微控制器的总体结构，它还要加上允许其与其需要的外部信号相连接的外围设备。这些外围设备可能包括数字、模拟的输入/输出、计数以及定时单元。本书的后续章节还会讲到其他更复杂的功能。例如，微控制器启动时需要的电子电路，以及驱动内部逻辑电路的时钟信号（在一些控制器中，时钟信号在内部产生）。

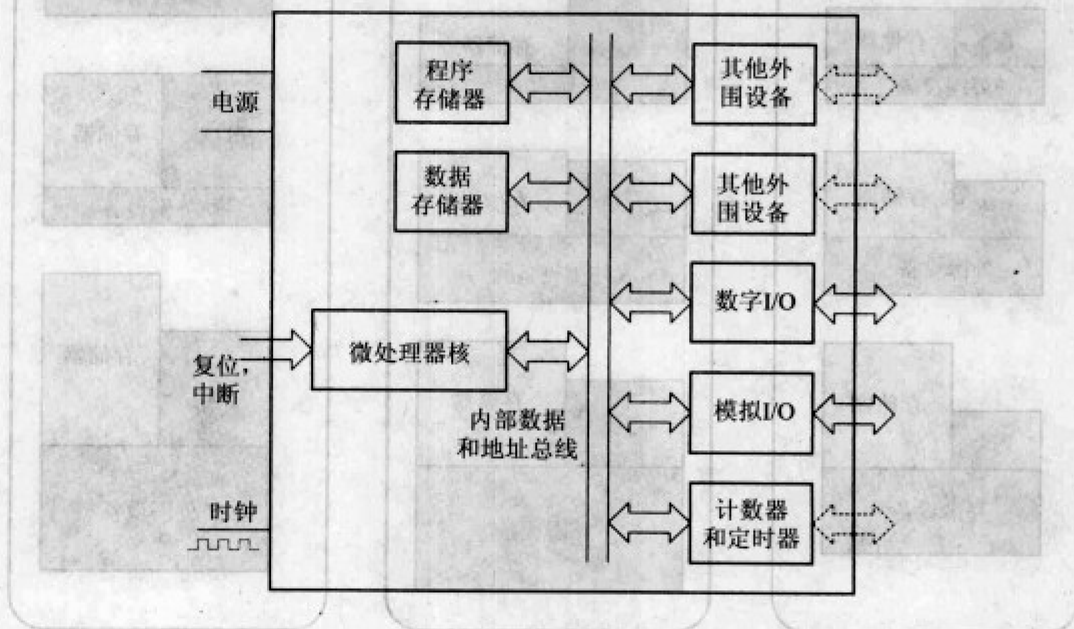


图 1-8 一般微控制器

1.4.3 微控制器系列产品

当前微控制器的类型有成千上万种，它们由众多不同的制造商生产。无论如何，所有这些微控制器肯定都是采用图 1-8 中的结构。制造商围绕固定的微处理器核创建微控制器系列。采用同样的微处理器核，加上不同的外围设备组合和不同的存储器大小，生产出同一系列的不同微控制器。图 1-9 象征性地体现了这种情况，制造商有 3 个微控制器系列，每个系列有自己的微处理器核。其中一个系列的微处理器核处理能力是 8 位系统，另一个系列的核为 16 位系统，还有一个系列的核为复杂的 32 位系统。给每个核加上不同的外围设备组合和不同的存储器大小，就可以得到大量该系列微控制器。由于同一系列的所有微控制器的核是固定的，所以指令集也是固定的，用户可以毫不费力地从同一系列的一个微控制器转到另一个微控制器。

图 1-9 只表示了每个系列中的一部分微控制器，并不是实际情况，每个系列都可能有超过 100 个微控制器，各个微控制器的性能都略有差异，而且某些微控制器是针对非常特定的应用而设计的。

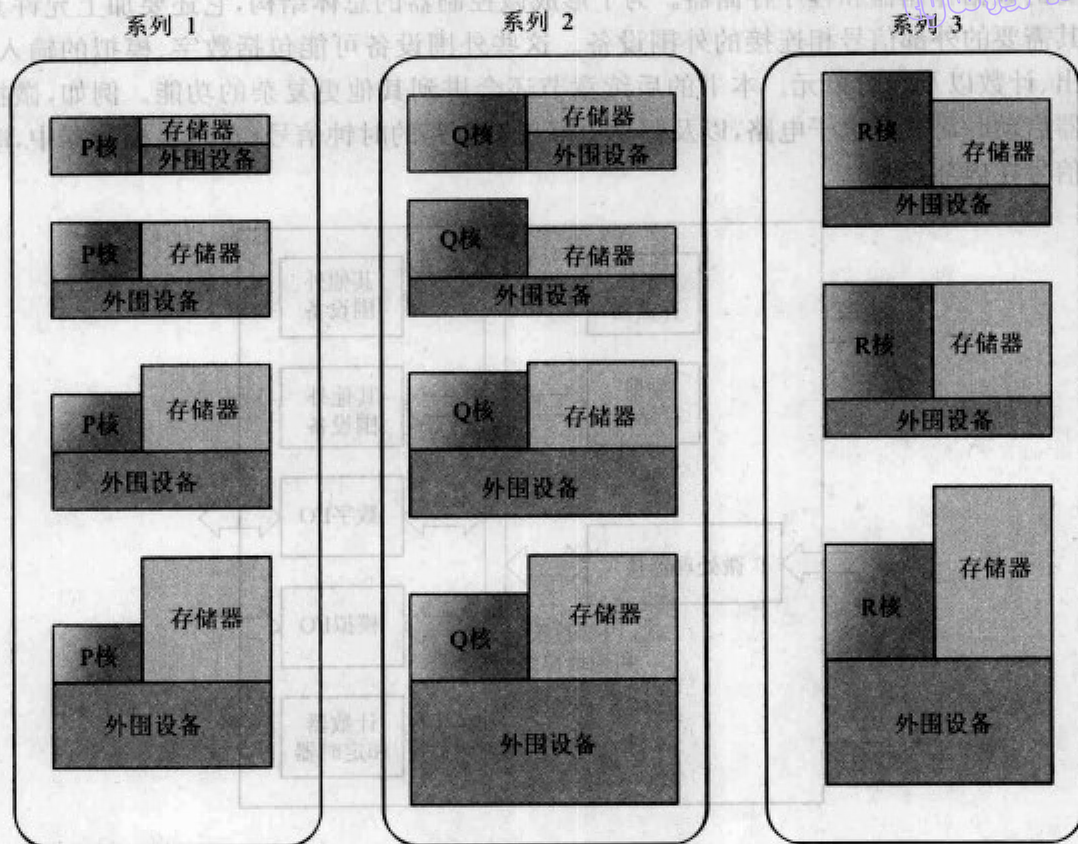


图 1-9 某厂商的微控制器组合

1.4.4 微控制器的封装和外观

集成电路制造有多种不同的形式，通常采用塑料或陶瓷作为封装材料。通过封装上的引脚与外部世界互连。在可能需要的地方，微控制器应当在物理上做得尽可能小，那么我们要问的是：是什么决定微控制器的大小？有趣的是，通常在微控制器中，决定整个微控制器大小的并不是集成电路芯片本身的大小，而是集成电路上互连引脚的个数和间距。

所以，值得仔细考虑的是微控制器中这些引脚携带了什么信息。需要强调的一点是，微控制器通常有密集的输入/输出。那么合理的想法是，微控制器需要有大量的引脚用于输入/输出，它还必须接电源和接地。对于后面即将学到的嵌入式系统，微控制器需要将所有存储器都放置在芯片上，这是一个合理的想法。因而，该类微控制器不需要早期微控制器所需要的与外部数据和地址总线相连的大量引脚。但是，传送程序信息到存储器以及为程序执行提供外部电源的互连引脚是必需的。所以，微控制器通常需要连接时钟信号、复位信号，还可能有中断输入。

图 1-10 中选择了一些微处理器和微控制器，说明了封装和大小的多样性。最右边

比其他微控制器大很多的微控制器是有 64 个引脚的 Motorola 68000。这是一款很早的微控制器了，采用的是双列直插式封装(Dual-in-Line Package, DIP)，即它的引脚沿着 IC 的最长边排列成两行，引脚间距为 2.54mm。因为 Motorola 68000 依靠外部存储器，它的许多引脚被指定数据和地址总线功能，这导致了这款微控制器的尺寸较大。右边第二个微控制器的年代相对较近，它是有 40 个引脚的 PIC 16F877。从外形上看，它与 Motorola 6800 相似，但实际上它们的引脚功能有很大的差别。由于采用了片上程序和数据存储器，它就不再需要外部数据和地址总线了。现在这些大量的引脚有更好的用途了，可以接入大量的数字输入/输出和其他信号线。中间的微控制器是有 52 个引脚的 Motorola 68HC705，它封装在方形的陶瓷里，陶瓷上开了一个窗口，用于擦除芯片上的可擦编程只读存储器(Erasable Programmable Read-Only Memory, EPROM)中的信息。这款微控制器的引脚间距为 1.27mm，所以虽然它的引脚数仍然很高，但整个 IC 的尺寸比 Motorola 68000 更紧凑。在它左边的是有 28 个引脚的 PIC 16C72。这款微控制器也有 EPROM 程序存储器，这样它也封装在双列直插式陶瓷里。最左边的是微小的采用表面安装的 PIC 12F508，它只有 8 个引脚。在它右边是有 18 个引脚的 PIC16F84A。

14

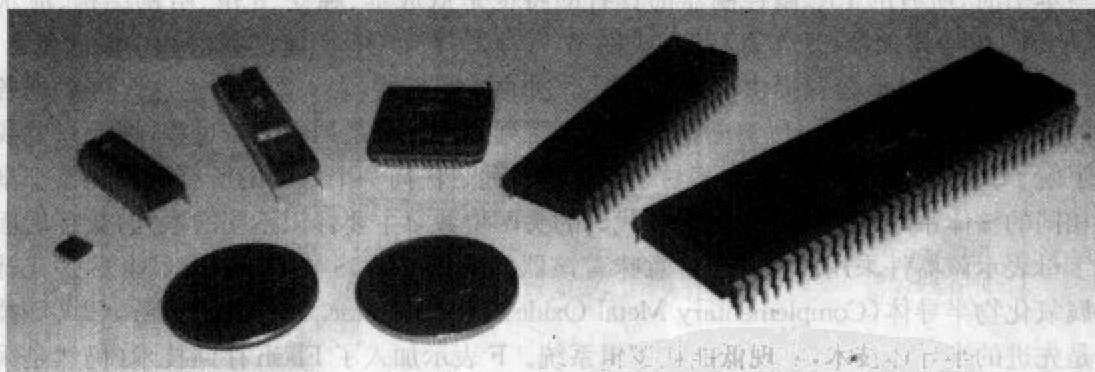


图 1-10 新老皆有的微处理器和微控制器。从左到右分别是：PIC 12F508, PIC 16F84A, PIC 16C72, Motorola 68HC05B16, PIC 16F877, Motorola 68000

1.5 Microchip 公司和 PIC 微控制器

1.5.1 背景

PIC 最初是由 General Instruments 公司设计，设计的目的是为了实现简单的控制应用，因此它被命名为外围设备接口控制器(Peripheral Interface Controller)。在 20 世纪 70 年后期，General Instruments 公司生产了 PIC®1650 和 1655 处理器。虽然设计相对粗糙且不正规，但它是一个完整独立的系统，并且拥有了一些重要的前瞻性特征。它的 CPU 非常简单，采用的是 RISC 结构，有一个单一的工作寄存器(working register)，且只有 30 条指令。与同时代的其他微处理器相比，输出引脚能够驱动更多的电

流源(current source)和电流阱(current sink)。[①] 此时,PIC 的特征已经出现:简单、独立、高速和低成本。

11

General Instruments 公司将它的半导体分部卖给了一群风险投资家,这些风险投资家肯定已经意识到这些奇怪的小器件有着巨大的潜力。到 20 世纪 90 年,PIC 微控制器的使用范围越来越广,事实上在很多方面它们逐渐替代了许多这方面已使用很久的器件。大多数情况下,与和它竞争的器件相比,PIC 微控制器的运行速度更快,所需芯片组更简单且原型开发更迅速。和许多竞争者不同,Microchip 公司让自己的开发工具简单、便宜甚至免费,此外,他们仍坚守在 8 位的世界。尽管 PIC 微控制器已经有了很大的变化,但即使是在最近的 PIC 设计中,我们仍能发现早期 General Instruments 公司的微控制器的特征。

1.5.2 今天的 PIC 微控制器

纵观当今 PIC 微控制器的种类,会让所有人都不知所措。目前,针对不同的应用,有成千上万种不同的 PIC 微控制器。我们可以试着识别所有这些器件共同的特征。编写本书时,所有的 PIC 微控制器都具有的特征是低成本、独立、8 位、哈佛结构、流水线、RISC、单一累加器(工作寄存器,或称 W 寄存器)的,还有固定的复位和中断向量。

今天,Microchip 公司提供了 5 种主要系列的微控制器,表 1-2 总结了这 5 种系列各自的特征。从表中我们可以清晰地看到一个系列到另一个系列的演变过程,那么只要我们了解了其中一个系列,就会很容易了解其他系列。任何一个系列的所有微控制器都享有相同的核体系结构和指令集。器件代码的头两位数字主要标识不同的系列,数字后面的字母表示该器件采用的工艺。C 意味着该器件采用 CMOS 技术,这里,CMOS 是互补金属氧化物半导体(Complementary Metal Oxide Semiconductor,CMOS)的简称,CMOS 技术是先进的半导体技术,实现低能耗逻辑系统。F 表示加入了 Flash 存储技术(仍然是采用 CMOS 技术作为核心技术)。位于数字之后的 A(此表中没有)表示首次发布器件的升级技术。X 表示某个阿拉伯数字,可以有多个值,对整个引用的数字并不重要。

15

表 1-2 PIC 型号比较

PIC 型号	栈大小(字)	指令字长	指令个数	中断向量个数
12CXXX/12FXXX	2	12 位或 14 位	33	无
16C5XX/16F5XX	2	12 位	33	无
16CXXX/16FXXX	8	14 位	35	1
17CXXX	16	16 位	58,包括硬件乘	4
18CXXX/18FXXX	32	16 位	75,包括硬件乘	2(有优先级)

例如,16C84 为该器件的初始版本。后来加入了 Flash 存储技术,重新发布的版本

① 电流源产生电流,电流阱吸收电流。——译者注

为16F84。如果以后针对16C84进行某种更进一步的技术升级，那么发布的版本就为16F84A。

Microchip公司习惯给每个微控制器系列都取名。第一个系列(16C5XX)被叫做"基础"系列。由该系列发展得到的称为"中端"系列，该系列的器件代码以"16C"或者"16F"开头(且第四位数字不是5)。"中端"系列演化得到"高端"系列，该系列的器件代码以"17C"开头。随着其系列的进一步发展，微控制器涵盖了简单和先进的体系结构，虽然仍采用术语来命名，但微控制器系列的命名习惯已经失去了其显著性。为了简便，本书采用"12系列"、"16系列"、"18系列"等来标识上述PIC系列。下面，让我们依次来认识它们。

1. 16C5X系列

该微控制器系列最先由General Instruments公司的微控制器演化而来，它有早期PIC设计的所有核心特征。由于只有两级栈且没有中断，所以该系列在可开发程序和硬件复杂性上有很大的局限性。特别是由于没有中断，可包含的片上外围设备种类受到限制，因为绝大多数的外围设备使用中断加强它们与CPU的连接。16C5X系列也发布了含有Flash存储器的版本，版本代码为16F5X。该系列已经发展成熟，但它只含有有限数量的微控制器，且没有被Microchip公司放在重要的位置。

2. PIC 16系列

该系列也叫"中端"系列，是16C5XX系列的一个升级版，在16C5XX系列中加入了中断(虽然只是单一的中断向量)，并增加了栈的容量。指令集在16C5X的指令集之上进行了小量的扩展。"中端"系列的微控制器品种繁多，技术含量更高且有许多不同的外围设备。与之前的系列相比，它加入了许多外围设备和大量片上存储器，外形上更大一些，同时处理能力强且应用广泛。

3. 12系列

12系列的微控制器是真正为微型应用设计的，它封装在很小尺寸的IC中(例如，只有8个或14个引脚)。它们拥有简单的体系结构和与16C5XX相同的指令集，可以看作是16C5XX系列的"分拆"版本。尽管尺寸小，但12系列的微控制器有较为丰富的外围设备，包括模数转换器和EEPROM(Electrically Erasable Programmable Read-Only Memory，电可擦编程只读存储器)数据存储器。虽然12系列的品种很少，但因为尺寸小而倍受青睐，而且这种系列还有加入更多功能的空间。

4. 17系列

与16系列的任何器件相比，引入17系列确实提高了CPU性能。在保持RISC策略的同时，指令集的大小几乎翻倍并且指令字长增加到16位。这样，诸如阅读表或移动数据这样的在"中端"系列中难以实现的可编程活动在这里就变得非常简单了。在17系列中还可以实现硬件乘法器。"中端"系列中只有一个中断向量，并且经常超负荷使用，而在17系列中中断向量增加到4个。虽然比16系列的功能强大不少，但17系列微控制器的数目仍受限，Microchip公司似乎将精力集中在18系列上，在向提供更强

16

处理能力方面发展。

5. 18 系列

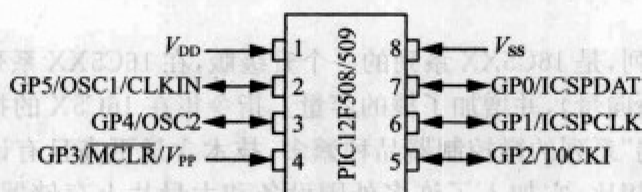
在本系列中，Microchip 公司开始认真关注一些先进处理器的发布。18 系列又扩展了指令集，将其增加到了 75 条，同时方便 C 语言编程。在某些版本中也有“扩展”指令集，包含一个更小的指令集合。这个系列的微控制器含有两个带优先级的中断向量。18 系列是一个处理能力非常强的微控制器系列，希望以后该系列会出现大量新的微控制器。

1.6　以 12 系列为例介绍 PIC 微控制器

作为最简单的 PIC 微控制器，12 系列非常适合作为 PIC 微控制器领域的入门系列。在本节学习到的控制器的特征会出现在更先进的 PIC 微控制器中，而在更先进的 PIC 微控制器中，除了这些特征，还会有一些更先进的特征。

我们将看到的是 PIC 12F508/509，图 1-11 为该微控制器的引脚连接图。508 和 509 的区别仅仅是后者有稍微大些的程序存储器和数据存储器。在刚开始学习时不必在意图中大多数引脚标签的意义——不用担心，后面会介绍它们的含义的。

16



对照表

V_{DD}：电源　　V_{SS}：接地

V_{PP}：可编程电压输入　　MCLR：主清零

OSC1，OSC2：振荡器引脚　　CLKIN：外部时钟输入

GP0到GP5：通用输入/输出引脚(除GP3外，其他都是双向的)

CSPDAT：In-Circuit Serial Programming™ 数据引脚

CSPCLK：In-Circuit Serial Programming™ 时钟引脚

图 1-11　PIC 12508/509 引脚连接图

17

在图 1-12 中可以看到微控制器的尺寸小得惊人。选择 12F508 作为简单的微控制器来达到介绍微控制器的目的，我们也必须意识到我们几乎正面临一个骗局：之前曾提到过微控制器应当有密集的输入/输出，而这里却只有 8 个输入/输出。那么我们要考虑的是：如果微控制器只有 8 个引脚与外界连接，那么它是怎样发挥作用的？在讨论微控制器的体系结构时，我们将试图对此做出回答。

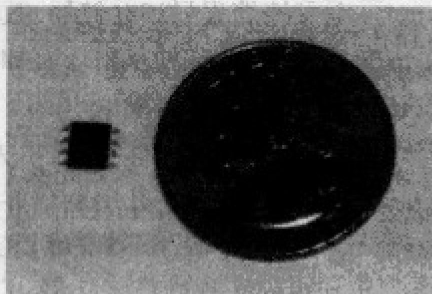


图 1-12　12F508 尺寸

12F508 体系结构

图 1-13 是 12F508 的注解框图，这可能是你所见的第一个 Microchip 公司微控制器的框图。它看起来复杂，但不必担心，我们会将它分解成便于理解的小部分。

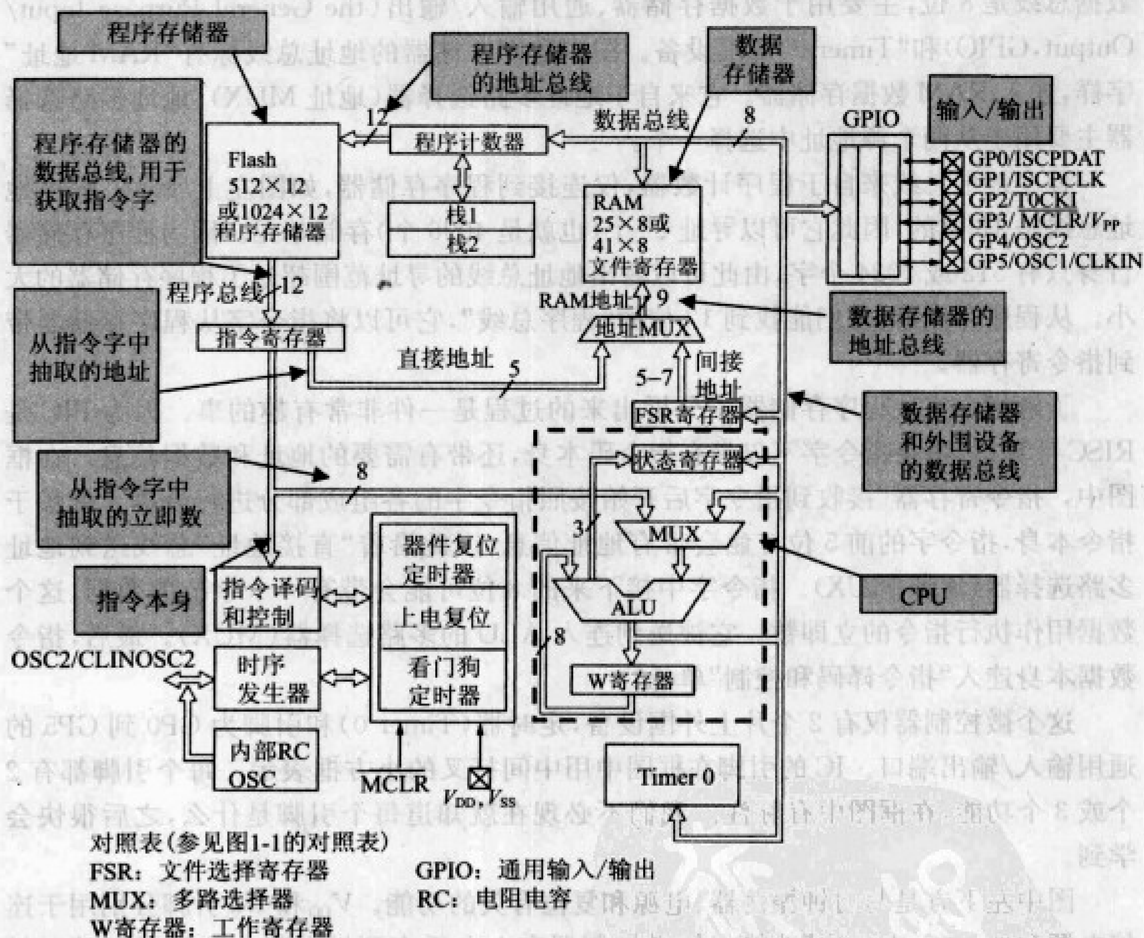


图 1-13 PIC 12F508/509 框图(阴影框中所附标签为作者所加)

让我们先找到图 1-8 所标识的微控制器的必要组成部分：处理器核(包含 CPU)、程序存储器、数据存储器(也就是 RAM)、数据通路和所有的外围设备。我们应当能够将部分特征与图 1-11 中的微控制器引脚联系起来。

虚线框围住的部分是 CPU，它由 ALU(算术逻辑单元)、工作寄存器(W 寄存器)和状态寄存器 3 个必要部分构成。状态寄存器有许多位，给出最近执行指令的结果信息。多路选择器(MUX)从两个源数据中选择一个送入 ALU。

18

508 的数据存储器仅 25B，509 的数据存储器有 41B。Microchip 公司在这里称 RAM 存储器区域为“文件寄存器”，有的地方也叫作“寄存器”。左上部为程序存储器，12F508 的程序存储器有 512 个 12 位的字，509 的程序存储器有 1024 个。

前面已经提到，PIC 体系结构的一个显著的特征是它为哈佛结构。因此我们能找到两条地址总线（一条属于程序存储器，一条属于数据存储器和所有外围设备）和两条数据总线（同样，一条属于程序存储器，一条属于数据存储器和所有外围设备）。在框图的右边我们很容易找到数据存储器和外围设备的数据总线它标有“数据总线”字样。数据总线是 8 位，主要用于数据存储器、通用输入/输出（the General Purpose Input/Output，GPIO）和“Timer 0”外围设备。图中数据存储器的地址总线标有“RAM 地址”字样，连入 RAM 数据存储器。它来自于地址多路选择器（地址 MUX），地址多路选择器主要用于从两个源地址中选择一个。

19

程序地址总线来自于程序计数器，仅连接到程序存储器，如图左上所示。程序地址总线是 12 位的，因此它可以寻址 2^{12} 个（也就是 4096 个）存储单元。因为程序存储器自身只有 512 或 1024 个字，由此可以看出地址总线的寻址范围超过了程序存储器的大小。从程序存储器，我们能找到 12 位的“程序总线”，它可以将指令字从程序存储器带到指令寄存器。

了解指令字从程序存储器中分拆出来的过程是一件非常有趣的事。因为 PIC 是 RISC 计算机，每条指令字不仅带有指令码本身，还带有需要的地址和数据信息。在框图中，“指令寄存器”接收到指令字后开始按照指令字的各组成部分进行分拆。依赖于指令本身，指令字的前 5 位可能会带有地址信息，因此沿着“直接地址”总线送到地址多路选择器（地址 MUX）。指令字中接下来的 8 位可能会带有一个字节的数据，这个数据用作执行指令的立即数。它被送到连入 ALU 的多路选择器（MUX）。最后，指令数据本身连入“指令译码和控制”单元。

这个微控制器仅有 2 个片上外围设备，定时器（Timer 0）和引脚为 GP0 到 GP5 的通用输入/输出端口。IC 的引脚在框图中用中间打叉的小方框表示。每个引脚都有 2 个或 3 个功能，在框图中有标注。我们不必现在就知道每个引脚是什么，之后很快会学到。

图中左下方是与时钟振荡器、电源和复位有关的功能。V_{DD} 和 V_{SS} 引脚分别用于连接电源和地。“上电复位”功能保持微控制器在上电后直到供电稳定这段时间处于复位状态。$\overline{MCLR}$ 输入用于把 CPU 置于复位状态，强制程序重新运行。内部时钟振荡器（内部 RC OSC）即使在没有外部引脚的情况下也可实现自身功能。外部振荡器能够通过 GP4 和 GP5 输入/输出引脚连接。振荡器信号用于调节“时序发生器”单元中的微控制器的性能。看门狗定时器是微控制器的安全特征，在程序失控时强制复位。

图 1-13 框图是图 1-8 中一般微控制器结构的详细描述，这对读者非常有益。然而在此并未对这些细节进行介绍，在后面的各章中详细介绍了微控制器的各部分。

1.7 其他微控制器——Freescale 微控制器

在 General Instruments 公司生产 PIC 1650 的同时，摩托罗拉公司致力于第一个 8

位微处理器——6800。后续非常优秀的微控制器都由这两个器件演变而来。除了继续做 8 位的器件外，摩托罗拉公司也相继开发了 16 位和 32 位的器件。然而，他们进而意识到尺寸小的 8 位微控制器的重要性，并且在 20 世纪 80 年代中期生产了一个 8 位微控制器 6805。该型号在之后的 20 年里不断地发展，并由它开发出了 68HC08 型号的微控制器。摩托罗拉公司的半导体部门在 2004 年进行了重组，更名为 Freescale。 [20]

Microchip 公司因为他们的非常小型的微控制器而为大众所知，Freescale 也已经开始在这一领域活动。图 1-14 从总体上显示了 Freescale 部门的 8 引脚微控制器，图 1-15 是该微控制器的简化框图。

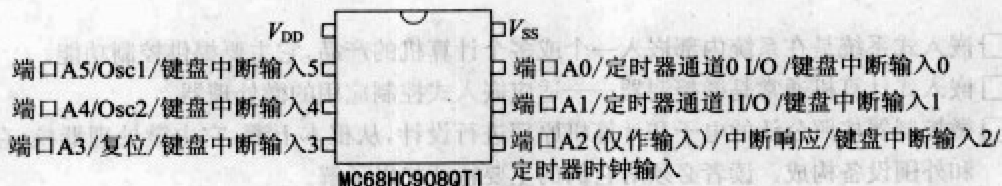


图 1-14 Freescale 的 68HC908 微控制器

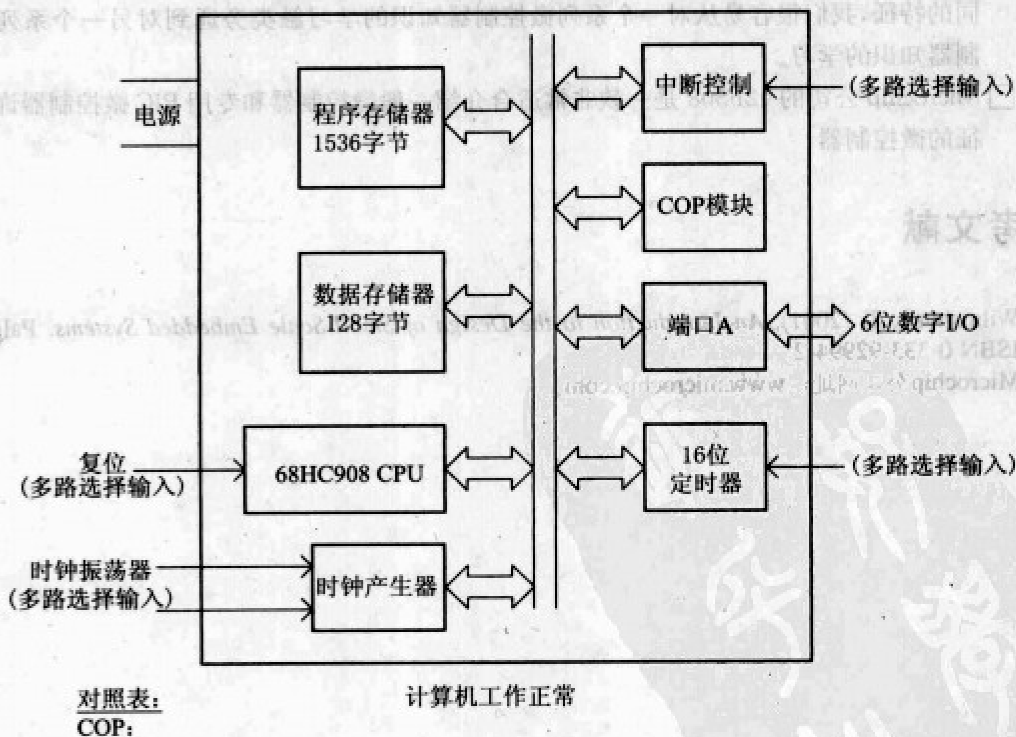


图 1-15 Freescale 的 MC68HC908QT1 微控制器——简化框图

虽然图并不详细，但仍可看出该微控制器采用了冯·诺依曼结构。相同的数据连接到数据存储器、程序存储器以及外围设备。数据总线是 8 位，也意味着程序和数据的字长都是 8 位。与 12F508 一样，Freescale 的这款器件也有片上程序和数据存储器， [21]

但体积稍大些。但是，需注意的是，F509 的存储器是由 12 位字而非字节来引述的。与 F508/9 的完全相同，Freescale 部门的器件也有 2 个外围设备、输入/输出端口和一个定时器。Freescale 部门的端口（端口 A）是 6 位，几乎直接等同于 12F508 的 GPIO。在可靠性上这两款器件有相同的特征，Microchip 公司采用看门狗定时器，Freescale 部门采用“计算机工作正常（Computer Operating Properly）”模块。这两款器件显著的不同之处是，即使是在这么小的微控制器上，Freescale 部门仍然在结构中加入了中断。

小结

- ❑ 嵌入式系统是在系统内部嵌入一个或多个计算机的产品，它主要提供控制功能。
- ❑ 嵌入式计算机通常是微控制器——适应嵌入式控制应用的微处理器。
- ❑ 微控制器按照公认的电子和计算机原理进行设计，从根本上看，它由微处理器核、存储器和外围设备构成。读者必须对它们的主要特征加以了解。
- ❑ Microchip 公司提供了很多微控制器，它们分为许多不同的系列。每个系列有相同的（或者非常相似的）中心体系结构和指令集。虽然这些微控制器各不相同，但它们都有一些共同的特征，我们很容易从对一个系列微控制器知识的学习触类旁通到对另一个系列微控制器知识的学习。
- ❑ Microchip 公司的 12F508 是一款非常适合介绍一般微控制器和专用 PIC 微控制器许多特征的微控制器。



参考文献

1.1. Wilmshurst, T. (2001). *An Introduction to the Design of Small-Scale Embedded Systems*. Palgrave. ISBN 0-333-92994-2.

22

1.2. Microchip公司网址：www.microchip.com。

第二部分　最小的系统和 PIC® 16F84A

本部分包括 5 章内容，以“小型”16 系列 PIC 微控制器为例介绍了微控制器的主要概念。本部分的重点是了解处理器核的体系结构，以及使用简单的外围设备。本章使用汇编语言编程，因为汇编程序更接近底层硬件。

第 2 章
PIC®16 系列和 16F84A

在第 1 章中，我们采用 12F508 作为介绍微控制器入门知识的器件，介绍了嵌入式系统并纵览了当下不同系列的 PIC®微控制器。现在我们将进一步详细学习 PIC 16“中端”系列。我们将使用 16F84A 作为该系列器件的例子，因为 16F84A 是该系列中相对较小的微控制器。学完 6 章内容之后，学习的重点将转移到 16F873A 上，16F873A 是“中端”系列中较大的微控制器。注意到'F84A 几乎包含于'F873A 中。所以，如果对 16F873A 更感兴趣，不必担心现在没学到 16F873A 的知识。在 16F84A 这个较小器件上学到的一切都可直接用于 16F873A。可以确定的是，我们在后续各章中学到的知识适用于所有 PIC 16 系列的微控制器。

我们将探究 16F84A 这个器件的整个体系结构，并讨论其存储器的一些细节，包括存储器技术和存储器映射。

因此，在本章中你将学到：

- ☐ PIC 16 系列概述；
- ☐ 16F84A 的整个体系结构；
- ☐ 16F84A 的存储系统，并回顾存储器技术；
- ☐ 16F84A 的其他硬件特征，包括复位系统。

如果你愿意，你还将学到：

- ☐ 微控制器结构的另一种方法，通过另一个微控制器系列的例子来介绍。

2.1 PIC 16 系列

2.1.1 PIC 16 系列概述

PIC 16 系列发展迅速，拥有各种各样几乎令人眼花缭乱的微控制器。因此，当我们谈论“系列”时，使用“扩展系列”这个概念，它有很多不同的微控制器。不过，16 系列仍维持这样的思想，那就是所有系列的微控制器有相同的处理器核和指令集，不同之处在于外围设备、其他实现的特征，以及封装大小。因此，16 系列符合图 1-9 的样式。

表 2-1 总结了一些 16 系列微控制器的细节特征，选取的微控制器都是我们在本书中会碰见的。虽然表 2-1 对了解整个 16 系列的特征仍有局限，但已包含了相当大的差异性。在 16 系列的扩展型号中，我们可以找到彼此之间联系紧密的微控制器的分组，表中的 16F84A 和 16F87XA 代表了这样的分组。16F84A 在表中第一个被列出来，我们将

在后面详细讨论它的特征。与它联系非常紧密的是16LF84A，16LF84A扩展了电源电压的范围，允许控制器在较低电压下工作。这些控制器都以不同的封装、不同的工作温度范围和不同的时钟频率范围实现。例如，16F84A有4MHz时钟和20MHz时钟两个版本。

26

表 2-1 一些 PIC 16 系列微控制器

设备编号	引脚数目*	时钟频率	存储器 (K=KB，即1024B)	外围设备/特性
16F84A	18	DC 至 20MHz	1K 程序存储器 68B RAM 64B EEPROM	1个8位定时器 1个5位并行端口 1个8位并行端口
16LF84A	同上	同上	同上	同上，扩展的电源电压范围
16F84A-04	同上	DC 至 4MHz	同上	同上
16F873A	28	DC 至 20MHz	4K 程序存储器 192B RAM 128B EEPROM	3个并行端口 3个计数器/定时器 2个捕捉/比较/PWM模块 2个串行通信模块 5个10位ADC通道 2个模拟比较器
16F874A	40	DC 至 20MHz	4K 程序存储器 192B RAM 128B EEPROM	5个并行端口 3个计数器/定时器 2个捕捉/比较/PWM模块 2个串行通信模块 8个10位ADC通道 2个模拟比较器
16F876A	28	DC 至 20MHz	8K 程序存储器 368B RAM 256B EEPROM	3个并行端口 3个计数器/定时器 2个捕捉/比较/PWM模块 2个串行通信模块 5个10位ADC通道 2个模拟比较器
16F877A	40	DC 至 20MHz	8K 程序存储器 368B RAM 256B EEPROM	5个并行端口 3个计数器/定时器 2个捕捉/比较/PWM模块 2个串行通信模块 8个10位ADC通道 2个模拟比较器

* 仅针对 DIP 封装

ADC：模数转换器(analog-to-digital converter)；PWM：脉宽调制(pulse width modulation)

第 2 章
PIC®16 系列和 16F84A

在第 1 章中，我们采用 12F508 作为介绍微控制器入门知识的器件，介绍了嵌入式系统并纵览了当下不同系列的 PIC® 微控制器。现在我们将进一步详细学习 PIC 16"中端"系列。我们将使用 16F84A 作为该系列器件的例子，因为 16F84A 是该系列中相对较小的微控制器。学完 6 章内容之后，学习的重点将转移到 16F873A 上，16F873A 是"中端"系列中较大的微控制器。注意到'F84A 几乎包含于'F873A 中。所以，如果对 16F873A 更感兴趣，不必担心现在没学到 16F873A 的知识。在 16F84A 这个较小器件上学到的一切都可直接用于 16F873A。可以确定的是，我们在后续各章中学到的知识适用于所有 PIC 16 系列的微控制器。

我们将探究 16F84A 这个器件的整个体系结构，并讨论其存储器的一些细节，包括存储器技术和存储器映射。

因此，在本章中你将学到：

- ☐ PIC 16 系列概述；
- ☐ 16F84A 的整个体系结构；
- ☐ 16F84A 的存储系统，并回顾存储器技术；
- ☐ 16F84A 的其他硬件特征，包括复位系统。

如果你愿意，你还将学到：

- ☐ 微控制器结构的另一种方法，通过另一个微控制器系列的例子来介绍。

2.1 PIC 16 系列

2.1.1 PIC 16 系列概述

PIC 16 系列发展迅速，拥有各种各样几乎令人眼花缭乱的微控制器。因此，当我们谈论"系列"时，使用"扩展系列"这个概念，它有很多不同的微控制器。不过，16 系列仍维持这样的思想，那就是所有系列的微控制器有相同的处理器核和指令集，不同之处在于外围设备、其他实现的特征，以及封装大小。因此，16 系列符合图 1-9 的样式。

表 2-1 总结了一些 16 系列微控制器的细节特征，选取的微控制器都是我们在本书中会碰见的。虽然表 2-1 对了解整个 16 系列的特征仍有局限，但已包含了相当大的差异性。在 16 系列的扩展型号中，我们可以找到彼此之间联系紧密的微控制器的分组，表中的 16F84A 和 16F87XA 代表了这样的分组。16F84A 在表中第一个被列出来，我们将

章中 PIC 12F508 的引脚连接图和总体框图相比有相似的地方，也有不同之处。从图 2-1中可看到 16F84A 有 18 个引脚，因此没有给每个引脚设计太多功能。例如，振荡器(引脚 15 和引脚 16)和复位(引脚 4——$\overline{\text{MCLR}}$)分别采用不同的引脚，这些引脚只提供其专一的功能。不过，和大多数微控制器相比，'F84A 仍是一个小型微控制器。



图 2-1 PIC 16F84A 的引脚连接图

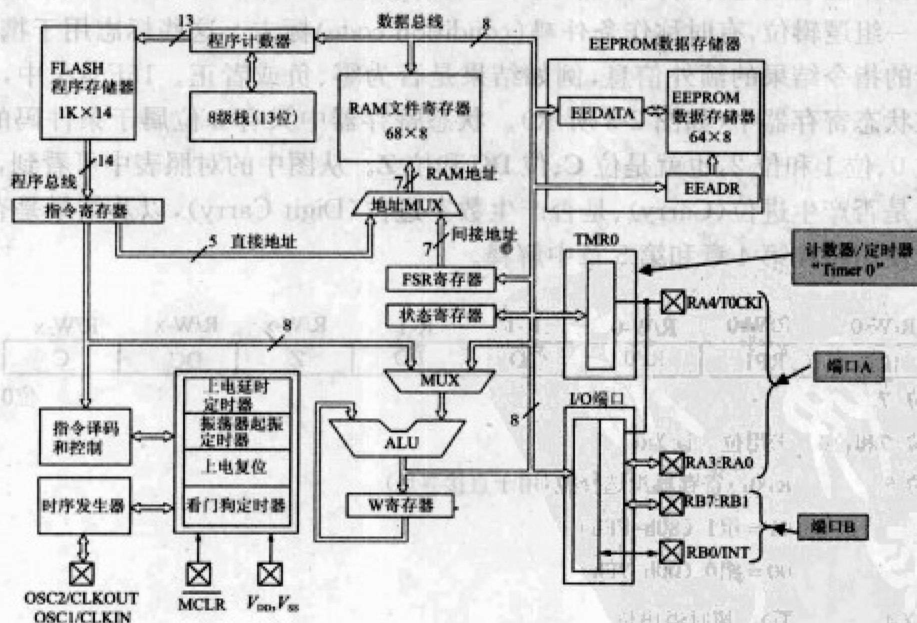


图 2-2 16F84A 框图(阴影框中所附标签为作者所加)

从体系结构上看，12F508 和 16F84A 很相似。事实上，前者包含于'F84A 中，它采用与 16F84A 几乎相同的 CPU、存储器、总线结构和计数器/定时器(TMR0)外围设备。但是首先要注意的是，为了满足整个 PIC 16 系列的需要，在微控制器中已经增加了地址总线的数量。作为 16 系列中较小的微控制器，'F84 不能充分利用这些新增部分。程序地址总线现在是 13 位，指令字是 14 位。所以，可寻址 2^{13}(即 8192)个存储单元。

然而，'F84 的程序存储器容量为 1KB，只占了 2^{13} 个可寻址单元的 1/8。但是，较大的地址总线大小对于较大的 16 系列器件是有用的，这在 16F876A 和 16F877A(见表 2-1)的程序存储器容量中可看到。RAM 的容量已经慢慢增加到 68 个存储单元，并且栈大小达到 8 个存储单元。

图中还显示了大量新增的重要部分。增加 EEPROM 存储器使得芯片具备了即使断电仍能保存数据的能力。现在有两个数字输入/输出端口，它们是有 5 个引脚的端口 A 和有 8 个引脚的端口 B。重要的是在这里增加了中断功能(我们将在第 6 章详细讨论中断)。从芯片外部看，中断输入在引脚 6 上，端口 B 的位 0 和外部中断输入共享该引脚。我们还可以看到有 3 个由外围设备产生的内部中断源。

总的来说，这款微控制器稍比 12F508 复杂，但却被证实带来较多变化且在小应用上非常有用。

状态寄存器

CPU 每次操作的结果都保留在工作寄存器中，但是这未必一定能提供刚刚执行的操作的所有信息。例如，如果一条加法指令的执行超出了 8 位的范围[①]，将会怎么样？工作寄存器不能指出这种情况，只能简单地保存错误的结果。因此，在所有 CPU 内部都构建了一组逻辑位，有时称作条件码(condition code)标志。这些标志用于携带有关最近执行的指令结果的额外信息，例如结果是否为零、负或者正。16F84A 中，这些标志保存在状态寄存器中(如图 2-3 所示)。状态寄存器中只有 3 位属于条件码的范畴，它们是位 0、位 1 和位 2，也就是位 **C**、位 **DC** 和位 **Z**。从图中的对照表中可看到，这 3 位分别表示是否产生进位(Carry)、是否产生数字进位(Digit Carry)，以及结果是否为零。这 3 位的使用将在第 4 章和第 5 章中解释。

R/W-0	R/W-0	R/W-0	R-1	R-1	R/W-x	R/W-x	R/W-x
IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C
位 7							位0

位 7和位6　未用位：读为0

位 5　RP0：寄存器组选择位(用于直接寻址)

01 = 组1（80h~FFh）

00 = 组0（00h~7Fh）

位 4　$\overline{TO}$：超时溢出位

1 = 上电（power-up）后，执行CLRWDT指令或SLEEP指令

0 = 发生WDT超时溢出

图 2-3　16F84A 状态寄存器

① 16F84A 是 8 位处理精度。——译者注

位 3　　PD：掉电位

1 = 上电后，或执行CLRWDT指令

0 = 执行SLEEP指令

位 2　　Z：零位

1 = 算术或逻辑操作结果为0

0 = 算术或逻辑操作结果非0

位 1　　DC：数字进/借位(ADDWF、ADDLW、SUBLW、SUBWF指令)(对于借位，极性相反)

1 = 有第3 位向第4 位进位或无第3 位向第4 位借位

0 = 无第3 位向第4 位进位或有第3 位向第4 位借位

位 0　　C：进/借位(ADDWF、ADDLW、SUBLW、SUBWF指令)(对于借位，极性相反)

1 =产生的结果的最高有效位(Most Significant Bit,MSB)向前有进位

0 =产生的结果的最高有效位向前无进位

注：减法通过加上第二个操作数的二进制补码来执行。对于移位指令(RRF, RLF)，源寄存器的最高位或最低位移入此位

图 2-3 (续)

2.3 存储器技术回顾

为了分析 16F84A 的存储器性能并用好嵌入式系统，有必要了解一些当前正在使用的存储器技术特征的知识。参考文献 1.1 的第 4 章详细介绍了存储器技术。接下来的几节简要总结了现今 Microchip 公司所使用的各种存储器技术。

28~29

理想的存储器读写时间短、随时保持数据、占据空间小且消耗能量低。实际上，没有任何存储器技术能满足所有这些理想的特征。通常，不同的存储器技术在这些特征上各占优势，有的在这方面强些，有的在那方面强些。没有哪种存储器技术是最好的，所以不同的存储器技术根据需要适用于不同的应用。

每个存储器都由一个存储单元阵列构成，阵列中每个单元保存数据的 1 位。单个单元的特征都可以反映整个阵列的特征，所以，本节按照存储器单元的设计来描述每种存储器技术。

2.3.1 静态 RAM

静态 RAM(Static RAM，SRAM)中，采用两个背靠背的晶体管对将每个存储单元设计成一个触发器(flip-flop)[①]。还有两个晶体管用于存储单元和主阵列的连接。只有在上电时数据才能保存，因此 SRAM 技术是易失性的。由于每个单元含有 6 个晶体

30

① 为了与 register 区别，本书中 flip-flop 称为触发器。——译者注

管,SRAM的密度不大[①]。然而,如果采用CMOS制造,SRAM将消耗非常少的能量,并且将它的数据保持至低电压(2V左右)。因而SRAM技术在采用电池供电的系统中用得很多。在微控制器中,SRAM主要用于数据存储器(RAM)。

2.3.2 EPROM

在EPROM技术中,每个存储单元仅由一个MOS晶体管构成,且每个MOS晶体管都有所不同。在每个晶体管中都嵌入了一个"浮栅(floating gate)"。采用热电子注入(Hot Electron Injection,HEI)技术,可以给浮栅充电。当浮栅没有充电时,晶体管工作正常且在被激活时存储单元输出代表一个逻辑状态。当浮栅充电时,晶体管不再正常工作且被激活时不响应,浮栅上的电荷完全陷于周围绝缘体的包围中。因此EPROM技术是非易失性的。但是,如果将EPROM暴露在强烈的紫外光下,它可以被擦除。浮栅上被捕获的电子可以从紫外光中获得能量,然后离开浮栅。

EPROM的一个特别版是"OTP"——一次可编程(One Time Programmable)。这个版本的EPROM采用塑料封装,没有窗口。因此,OTP只能被编程一次且永不能被擦除。

由于一个单元只有一个晶体管,EPROM的密度很大并且稳定性好。EPROM需采用石英窗口和陶瓷封装,以保证它能够被擦除,但这种需求会提高其价格并降低其灵活性。EPROM通常集成在许多微控制器中,用作程序存储器,由于EPROM需要采用石英窗口和陶瓷封装,因而迫使整个微控制器也要采用陶瓷封装和石英窗口(如图1-10所示)。作为存储器技术,EPROM如今正快速地被Flash取代,稍后马上会介绍Flash。

2.3.3 EEPROM

EEPROM也采用浮栅技术。EEPROM的结构尺寸更精细,因而它可以采用另一种方式对浮栅充电,那就是隧道效应(Nordheim Flower Tunneling,NFT)。采用NFT可通过电擦写存储单元。为了实现电擦写,需在存储单元周围布置大量的开关晶体管,因而EEPROM没有EPROM的高密度性。

通常,EEPROM可以一个字节一个字节地写入和擦除,这使得EEPROM特别适用于存储单条数据,例如电视设置或手机号码。写入和擦除都可以在限定的时间内完成,最多在数毫秒的时间之内完成,而读出能够在常规半导体存储器的访问时间(也就是在数微秒或者更少的时间)之内完成。和EPROM一样,因为浮栅上的电荷也完全陷于周围绝缘体的包围中,所以EEPROM是非易失性的。如今EEPROM结构非常精细,它会由于重复擦写而逐渐受损。所以,厂商通常会给定一个他们所生产的EEPROM存储器能保证完成的最少擦写次数。

---

① 单位面积能存储的数据量不多。——译者注

2.3.4 Flash

Flash 技术是浮栅技术的进一步发展。Flash 的每个存储单元仅有一个晶体管，同时采用 HEI 和 NFT 进行电擦写。Flash 没有 EEPROM 所带的额外开关晶体管，因而只在块内能擦除。所以，Flash 同 EPROM 一样具有非常高的密度。和 EEPROM 一样，Flash 也会由于重复擦写而逐渐受损，所以它不能无限制地写入和擦除。

除了不能一个字节一个字节地擦除外，Flash 存储器技术非常强大。Flash 现在是包括数码相机、“记忆棒”、笔记本电脑和微控制器程序存储器在内的很多产品的重要特征。

2.4 16F84A 的存储器

正如图 2-2 所示，16F84A 中最少有 4 个存储区域。表 2-2 总结了这 4 个存储区域。每个存储器都有自己独特的功能和访问方式。

表 2-2 16F84A 存储器特征

存储器功能	技术	尺寸	易失性/非易失性	特性*
程序	Flash	1K×14 位	非易失性	10 000 个擦写周期，典型值
数据存储器(文件寄存器)	SRAM	68 字节	易失性	保存数据电源电压低至 1.5V
数据存储器(EEPROM)	EEPROM	64 字节	非易失性	10 000 000 个擦写周期，典型值
栈	SRAM	8×13 位	易失性	

*信息来自于 16F84A 数据手册[2.1]。

2.4.1 16F84A 程序存储器

图 2-4 是 16F84A 的程序存储器映射。从图中我们可以看到，它包括 3 个部分：程序计数器(Program Counter)、栈和实际的程序存储器。这 3 个部分必须要一起工作才能发挥作用。程序存储器装载微控制器执行的程序代码。程序的形式是一串指令，而程序计数器保存微控制器将要执行的下一条指令的地址。所以，程序计数器就像指向程序存储器的指针，如图中所示。当调用子例程或产生中断时，程序计数器的值被保存在栈中。图中列出的指令有 **CALL**、**RETURN**、**RETFIE** 和 **RETLW**，这些指令都与子例程和中断相关。如果你现在不知道它们的含义，不必担心，我们将在后续各章中学习。

从图 2-4 中我们可以看出程序存储器的地址范围为 0000～03FFh[①]。由于采用 13 位的程序计数器，该微控制器理论上可寻址的范围为 0000～1FFFh。03FFh～IFFFh 的地址空间在这里并没有用到，如图 2-4 中灰色部分所示。

① 十六进制表示。——译者注

程序存储器最开始的位置标有复位向量(reset vector)。当程序第一次开始运行时，例如在上电时，程序计数器被设置为 0000。所以，程序计数器指向的最初存储位置是复位向量。因此，程序员必须把他的第一条指令放在这个位置。对于中断服务程序，外围设备中断向量(peripheral interrupt vector)起着和复位向量相似的作用，我们将在第 6 章中学习这些内容。

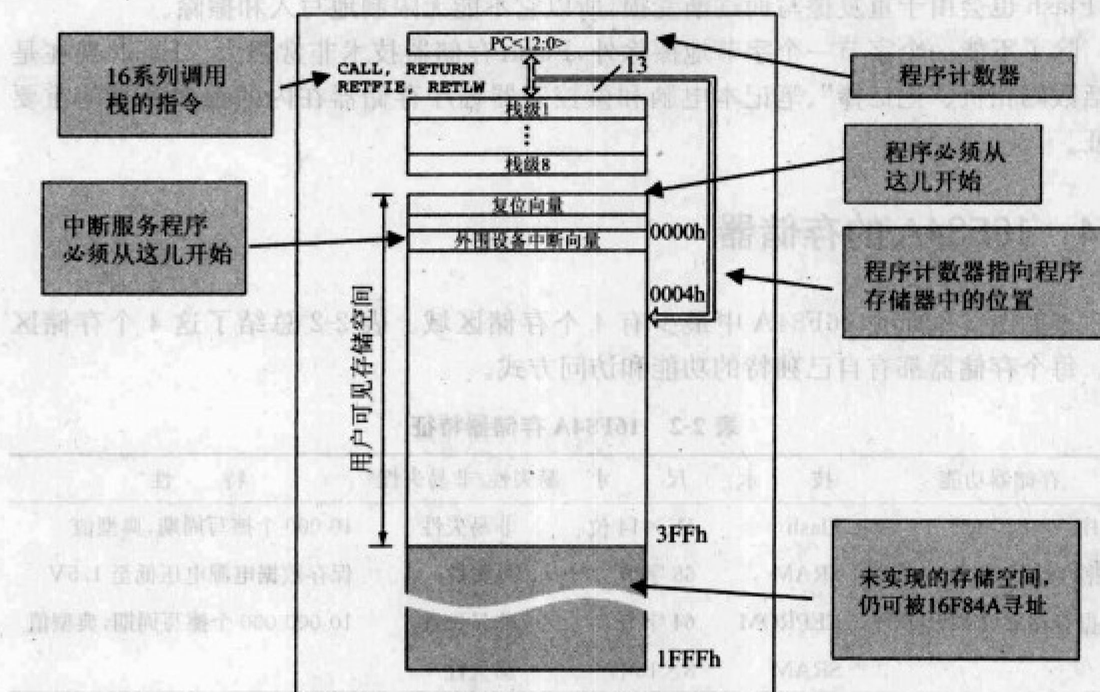


图 2-4 16F84A——程序存储器和栈(阴影框中所附标签为作者所加)

2.4.2 16F84A 数据和特殊功能寄存器存储器(RAM)

图 2-5 是 RAM 的存储器映射。存储器区域按存储区划分，它被划分为 2 个重要的区域。第一个区域为通用数据存储器，它占据的位置为 0Ch～4Fh。在它上面是特殊功能寄存器(Special Function Register，SFR)。下面我们开始讨论刚刚提到的这两个生疏的概念。

1.“存储区划分”寻址

所有存储器空间都存在这样一个问题，存储器越大，地址总线也必须越大。避免地址总线过大的一个方法是将存储器划分为许多较小的模块，这些模块称为存储区(bank)，它们都有相同的大小。这样就可以使用较小的地址总线，它可以以相同的方式访问所有的存储区，无论何时这些存储区中仅有一个被识别作为地址指定的目标。

PIC 微控制器在 RAM 中采用存储区划分的结构，16F84A 仅有 2 个存储区。每个存储区的地址为 7 位 RAM 地址，如图 2-2 所示。活动存储区由状态寄存器(图 2-3)的

位 5 选择。程序员在访问存储器之前必须确保状态寄存器中的存储区位设置正确。

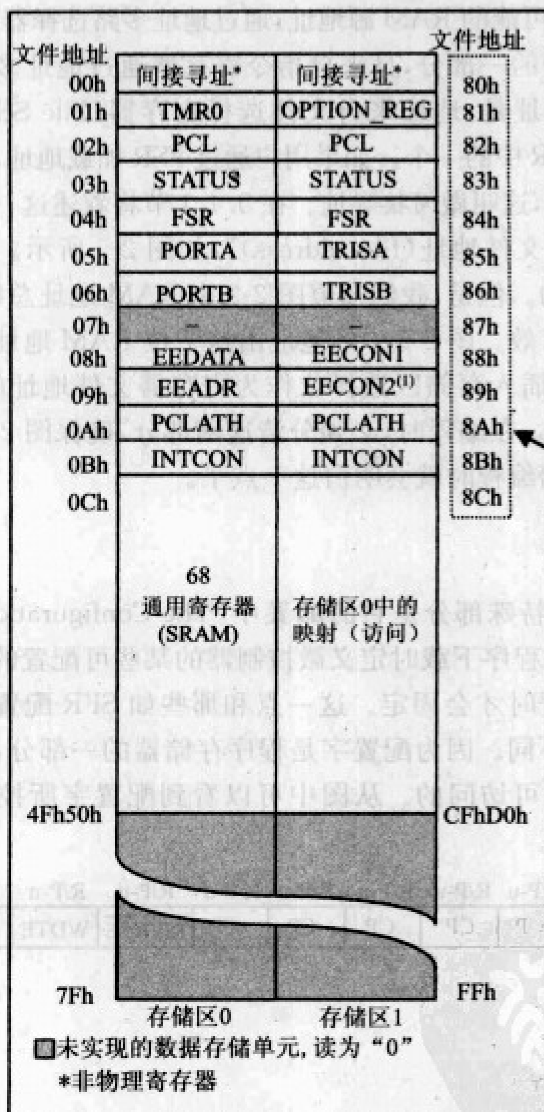


图 2-5 16F84A 的数据存储器和特殊功能寄存器映射(阴影框中所附标签为作者所加)

2. 特殊功能寄存器

SFR 是 CPU 和外围设备之间交互的途径，我们将在后面详细地介绍它。对于 CPU 来说，SFR 多多少少更像一个普通的存储器单元——你可以经常读写它。它之所以"特殊"是因为该存储单元的每一位都有双重功能，每一位都连线到微控制器的某一个外围设备上。这些位要么用于设置外围设备的运行模式，要么在外围设备和微控制器核之间传送数据。在我们开始学习 16F84A 的外围设备时，也要学习图 2-5 中的每个 SFR。注意，在图 2-2 中有 4 个 SFR，你能找到它们吗？

3. RAM 寻址

从图 2-2 中可以看到有 2 个可能的 RAM 源地址，通过地址多路选择器(地址 MUX)从中选择一个。如果地址是指令的一部分，且来自指令寄存器通过地址多路选择器路由，这叫做**直接寻址**。另一个寻址是，地址来自文件选择寄存器(File Select Register，FSR)，在图 2-5 中可发现它是 SFR 中的一个。如果用户通过 FSR 加载地址，那么 FSR 就可以作为数据存储器的地址使用，这叫做**间接寻址**。在 5.4.1 节将叙述这一点。

实际的存储器地址标记为“文件地址(file address)”，如图 2-5 所示。至少从右边栏中可以看到这些地址是 8 位的。但是，我们知道图 2-2 中 RAM 地址总线只有 7 位，如果采用直接寻址还只有 5 位有效。图 2-5 中的地址由这 7 位 RAM 地址总线和状态寄存器中的存储区选择位组成，插入存储区选择位作为寄存器文件地址的第 8 位，也就是最高有效位，这一点很重要。在编程时，必须分清这两部分，确保图 2-5 中的 MSB 用于存储区选择位。当我们开始编程时就会明白这一点了。

2.4.3 配置字

16F84A 程序存储器的一个特殊部分是它的**配置字**(The Configuration Word)，如图 2-6 所示。配置字允许用户在程序下载时定义微控制器的某些可配置的特征。所配置的特征直到微控制器下次编程时才会固定。这一点和那些如 SFR 配置一样在正常程序控制下的可选择特征截然不同。因为配置字是程序存储器的一部分，在程序内或者程序在运行时，配置字都是不可访问的。从图中可以看到配置字所控制的实际特征，这会在本章和后续章中讲解。

R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u
CP	CP	CP	CP	CP	CP	CP	CP	CP	CP	$\overline{\text{PWRTE}}$	WDTE	F0SC1	F0SC0

位13　　　　位0

位13~4　**CP**：代码保护位
1=代码保护关
0=所有程序都进行代码保护

位3　$\overline{\textbf{PWRTE}}$：上电定时器使能位
1=上电定时器禁止
0=上电定时器使能

位2　**WDTE**：看门狗定时器使能位
1=WDT禁止
0=WDT使能

位1~0　F_{OSC1}:F_{OSC0}：振荡器选择位
11=RC振荡器
10=HS振荡器
01=XT振荡器
00=LP振荡器

图 2-6　16F84A 配置字

2.4.4 EEPROM

EEPROM 是非易失性的，且对保存可能改变的数据变量特别有用，但可能要长期作为保存数据的媒介。有关 EEPROM 的例子有 TV 调谐器的配置、手机中电话号码的存储，以及测量仪器上的刻度设置。

在 16F84A(甚至任何 PIC 微控制器)中，EEPROM 不在主要的数据存储器映射中。相反，它通过 **EEADR** 寄存器寻址，通过 **EEDATA** 寄存器传送数据，如图 2-2 右上部所示。从图 2-5 中可以看出，这两个寄存器都是 SFR。

正如前面介绍存储器技术的部分所指出的，读 EEPROM 是一个简单的过程，但写 EEPROM 却不是。写 EEPROM 在电子运动上花费大量的时间(也就是数毫秒)[①]，那么必须要非常小心以避免不必要的写入。因此，需要一组控制去开始读写过程，并检测写入何时结束。寄存器 **EECON1** 的各位就是这组控制(如图 2-7 所示)。读一个 EEPROM 单元时，需要的读地址必须放在 **EEADR** 中且设置 **EECON1** 中的 **RD** 位。然后将这个存储单元的数据复制给寄存器 **EEDATA**，这样就能立即读出。写一个 EEPROM 单元时，需要的写数据和写地址必须分别放在 **EEDATA** 和 **EEADR** 中。在 **WREN** (写

35

U-0	U-0	U-0	R/W-0	R/W-x	R/W-0	R/S-0	R/S-0
—	—	—	EEIF	WRERR	WREN	WR	RD

位7 位0

位7~位5 未用：读作"0"

位4 **EEIF**：EEPROM写操作中断标志位
1=写操作完成(必须在软件中清零)
0=写操作未完成或还未开始

位3 **WRERR**：EEPROM错误标志位
1=写操作过早中止(在正常操作下，每次MCLR复位或WDT复位)
0=写操作完成

位2 **WREN**：EEPROM写使能位
1=允许写周期
0=禁止写EEPROM

位1 **WR**：写控制位
1=开始写周期。一旦写完成，WR位通过硬件清零。WR位只能在软件中设置(不能清零)
0=EEPROM写周期完成

位0 **RD**：读控制位
1=开始读EEPROM。RD在硬件中清零。RD位在软件中只能设置(不能清零)
0=未开始读EEPROM

图 2-7 EECON1 特殊功能寄存器(地址 88_H)

---

① 写 EEPROM 的过程实际上是对晶体管寄生电容充电的过程，电子运动使电容充电。——译者注

使能)位设为高时,写过程启动。写过程之后,依次将字节 55_H 和 AA_H 写到寄存器 EECON2 中。对这些代码的固定要求能帮助确保不会发生不必要的写入,例如在上电或掉电时。然后将 WR 位设为高,这时写过程实际上才执行。写过程的完成通过设置 EECON1 中的 EEIF 位来标记。

2.5 一些有关时序的问题

2.5.1 时钟振荡器和指令周期

任何微处理器或微控制器都是由时序电路和组合逻辑电路组成的复杂电子电路。微处理器或微控制器以让人难以置信的速度通过一系列的复杂状态来按序执行,每个状态都与它执行的指令序列有关。该过程的详细情况对我们是隐藏的,但仍需提供"时钟"信号,"时钟"信号是一个连续运行有固定频率的方波。微处理器运行的整个速度完全决定于该时钟频率。不是只有 CPU 才依赖于时钟。在大多数微控制器中,从计数器/定时功能到串行通信,许多必要的与时序相关的功能也来自于时钟。此外,微控制器的整个功耗很大程度上也取决于时钟频率,高速运行比低速运行需要更多的能量。

如表 2-1 所示,每个微控制器有指定的时钟频率范围。由设计者决定所需的时钟频率和选择生成时钟源的方法。由于很多方面取决于时钟频率和时钟频率的稳定性,决定时钟频率和生成时钟源方法是一项非常有挑战性的工作。我们将在第 3 章进一步讨论这一点。

在任何一个微处理器内,一个固定的数值直接除以主时钟信号可以得到一个较低频率的信号。这个较低频率的信号的每个周期叫做机器周期(machine cycle)或指令周期(instruction cycle)。Microchip 公司使用后"指令周期"这个术语。指令周期是在处理器运行中的基本时间单位,例如用作衡量一条指令的执行需多长时间。原始时钟保留以产生指令周期内的相位或时间段。PIC 16 系列微控制器中,4 除以主振荡器信号得到该系列微控制器的指令周期时间。

表 2-3 给出了常用的时钟频率和由它们得到的指令周期长度。对于最快的时钟频率(即 20MHz),指令周期频率为 5MHz,周期为 200ns。微控制器稍便宜点的版本(16F84-04)的最大时钟频率为 4MHz,它在此频率下的指令周期为 1μs。正如接下来将看到的,这是在使用了软件延迟循环和计数器/定时器的简单时序应用范畴中的惯用值。包括手表在内的超低能耗应用的常用时钟频率是 32.768kHz,其指令周期为 122.07μs。因为频率低,所以消耗的能量非常少,但严格上说不能进行高速计算。

36~37

表 2-3 PICl 16 系列的不同时钟频率下的指令周期

时钟频率	指令周期	
	频　　率	周　　期
20MHz	5MHz	200ns
4MHz	1MHz	1μs
1MHz	250kHz	4μs
32.768kHz	8.192kHz	122.07μs

2.5.2 流水线操作

PIC 微控制器所采用的 RISC 指令集和哈佛存储器映射相组合的结构还有一个好处：指令可以流水(pipelined)① 。计算机的程序存储器中的每条指令先被取出然后再被执行。在许多 CPU 中，有两个需要相继执行的步骤——首先 CPU 取出指令然后执行指令。然而，如果程序存储器有自己的与数据存储器分离的地址和数据总线(也就是哈佛结构)，那么 CPU 没有理由不设计成这样——当 CPU 正在执行一条指令时，它已经在取下一条指令了。这就叫做流水线操作(Pipelining)。如果取指周期和执行周期总是一样长，流水线操作就会达到最好的效果。例如，RISC 结构中有同样长的取指周期和执行周期。这种很简单的改进设计可以使执行速度提高一倍。

所有的 PIC 微控制器都实现流水线操作，其中一个原因是它们有相对较高的运行速度。每条指令在上一条指令正在执行时被取出。流水线操作仅对能造成程序计数器值改变的指令失效，例如，程序分支或跳转指令。在这种情况下，获取的指令不再是程序所需的指令②。那么流水线操作必须重新开始，并在时间上有一个指令周期的损失。

图 2-8 中的框图表示了 16 系列微控制器中的流水线操作过程。在图中可以看出，

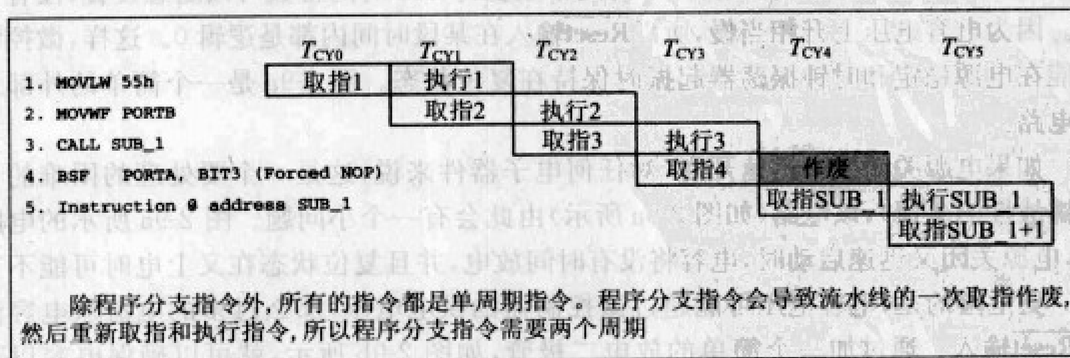


图 2-8 指令的流水线操作

① 即以流水线方式执行。——译者注

② 由于 PC 值的改变，要获取的指令不再是顺序流水的指令，而是跳转之后的指令。——译者注

在执行指令 1 时，指令 2 已经取出，同样当指令 2 正在执行时，指令 3 已经取出，以此类推。图左边是一个指令序列例子。然而在看懂该图时，除了知道 **CALL** 指令造成程序分支外，不必懂得其他指令的含义。在执行指令 3 时，**CALL** 指令(指令 3)后的指令 4 取出。但是，由于程序分支的原因，指令 4 不再需要，而在新指令取出时，不得不损失一个周期的时间。

2.6 上电和复位

当微控制器上电时，它必须从程序起始处(对于 16F84A 来说，也就是它的复位向量，如图 2-4 所示)开始执行。只有在内置的显示电路检测出器件上电并将程序计数器强置为 0 后，程序才开始执行。除此之外，设置 SFR 也非常有用，使得外围设备能初始化在一个安全和禁止工作的状态。这个“准备启动”的状态叫做复位(Reset)。CPU 在离开复位状态时开始运行程序。

38

16F84A 中有一个复位输入，$\overline{\text{MCLR}}$(“主清零”，在引脚 4 上，如图 2-1 所示)。只要复位输入保持在低，微控制器就保持在复位状态。当复位输入设置为高时，程序开始执行。如果在程序运行时，该引脚设置为低，那么程序执行立即停止且微控制器被强制回到复位模式。

仍然存在一个问题，那就是程序执行实际上应当何时才允许开始。对嵌入式系统来说，上电时是一个危险的时刻。电源和时钟振荡器都需要一定的时间去稳定，且在复杂的系统中，对电路不同地方供电也需要不同的时间来变稳定。显然，这种情况需要相当细心地处理。程序执行是怎样延迟到供电已经稳定时才开始的呢?

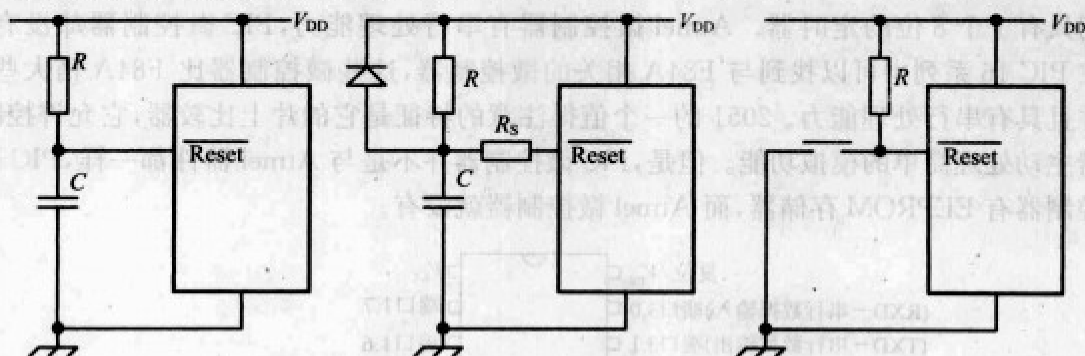
图 2-9 是解决“在上电时我们做些什么”这个问题的一个简单方法，图中也举例说明了每个微控制器都有一个有效的低复位输入。如果电阻电容电路连接到复位输入，那么当上电时，电容电压会按照 RC 时间常数上升，RC 时间常数可以随意设置，没有上限。因为电容电压上升相当慢，所以$\overline{\textbf{Reset}}$输入在某段时间内都是逻辑 0。这样，微控制器能在电源稳定和时钟振荡器起振时保持在复位状态。图 2-9a 是一个简单的外部复位电路。

如果电源关闭后又迅速开启(对任何电子器件来说，这是一个要处理的困难的且有挑战性的事情)，该电路(如图 2-9a 所示)由此会有一个小问题。图 2-9a 所示的电路中，电源关闭又迅速启动时，电容将没有时间放电，并且复位状态在又上电时可能不正常。更危险的是，电容电压可能超过微控制器的供电电压，那么过多的电流从电容流向$\overline{\textbf{Reset}}$输入。通过加一个简单的放电二极管，如图 2-9b 所示，就可以确保电容以和 V_{DD}电源差不多的速率放电。如果电容电压无意中超过微控制器的电源电压或者发生其他错误状态，电阻 R_S也用于限制电流流向$\overline{\textbf{Reset}}$输入。

39

如果设计者希望在电路中包含一个复位按钮，那么可以使用图 2-9c 中所示的电路。这个电路对原型电路非常有用，因为原型电路中会存在大量的测试。这样可以方

便地为可能崩溃的程序复位。R 是一个上拉电阻，R 的取值范围为 10kΩ～100kΩ。在商业器件中，通常不要求有复位按钮。这里只是将产品设计成用户可以复位，这样的产品在商业器件中是不需要的。



(a) 上电复位，最简单的　(b) 上电复位，有放电二极管和保护电阻　(c) 用户复位按钮

图 2-9　外部复位电路——常见的带 $\overline{\text{Reset}}$ 输入的微控制器

Microchip 公司的一个目标是将他们的微控制器需要的外部组件个数减到最少，而图 2-9 的组件恰恰属于这类外部组件。因此，16F84A 包含了相当复杂的片上复位电路，在很多情况下，图 2-9a 和图 2-9b 中所示的电路组件是多余的。片上含有上电延时定时器(PWRT)，用户可以通过设置配置字的第 3 位使能它(如图 2-6 所示)。16F84A 检测到微控制器已经上电，然后上电延时定时器使微控制器在复位状态维持一段固定时间。一旦这个维持动作结束，微控制器离开复位状态，程序开始执行。实际上，如果电源电压上升非常慢，图 2-9b 中所示的电路才会用到。在 2.8 节中将更详细地讨论上电延时定时器和内部复位电路更深层次的细节。

那么如果我们不想使用 $\overline{\text{MCLR}}$ 输入时，我们该对它做些什么呢？必须知道的是，$\overline{\text{MCLR}}$ 输入禁止悬空，什么也不接。最简单的做法是将该输入接到电源线上，然后就不用管它了。

2.7 其他微控制器——Atmel AT89C2051 微控制器

Atmel 公司生产小型(非微小的)微控制器，与之相比，PIC 16 系列器件在尺寸上更小一些，但 Atmel 公司的微控制器只比 16F84A 稍大一点。Atmel 的微控制器是基于 8051 处理器核，8051 处理器核最初由 Intel 开发，之后被大量其他生产商采用，其中就包括 Atmel 公司和 Philip 公司。

40

图 2-10 总结了 Atmel AT89C2051 的一些特征[2.4,2.5]。AT89C2051 有 20 个引脚，因而比 16F84A 稍大一点，这反映在它的内部处理能力上。尽管 AT89C2051 和 16F84A 有不同之处，但关注它们的相似之处是非常有意思的。两者都有片上程序和数据存储器——用于存储程序的 Flash 和存储数据的 SRAM。晶振输入、复位和供电

电源也非常相似。PIC 微控制器有 13 个数字输入/输出引脚，Atmel 微控制器有 15 个数字输入/输出引脚，两者都有直接 LED 驱动能力。PIC 微控制器有 1 个外部中断，而 Atmel 微控制器有 2 个外部中断。Atmel 微控制器有 2 个 16 位定时器，PIC 微控制器则只有 1 个 8 位的定时器。Atmel 微控制器有串行处理能力，PIC 微控制器却没有。在 PIC 16 系列中可以找到与'F84A 相关的微控制器，这些微控制器比'F84A 稍大些，并且具有串行处理能力。'2051 的一个值得注意的特征是它的片上比较器，它允许控制器主动处理简单的模拟功能。但是，PIC 微控制器并不是与 Atmel 器件都一样，PIC 微控制器有 EEPROM 存储器，而 Atmel 微控制器就没有。

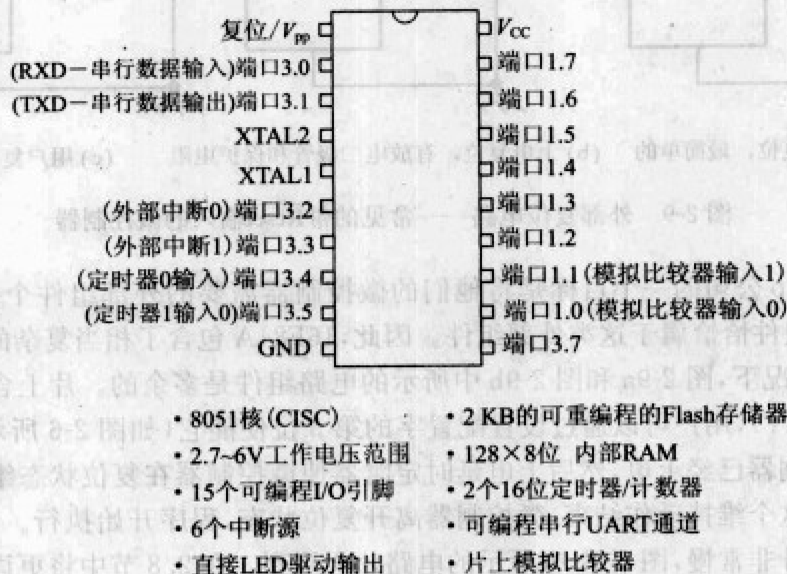


图 2-10 Atmel AT89C2051 引脚连接图和小结

然而，上述讨论主要是比较两者的外围设备，而生产商可以轻而易举地增加或减少这些外围设备。如果我们使用这两种微控制器，会发现什么别的不同呢？答案就在于处理器核和指令集。ATmel 微控制器是 CISC 器件，采用 8051 指令集，第 4 章我们会更进一步讨论该指令集，本节中仅关注 PIC 微控制器核的优点。本章前面已经提到，作为 RISC 处理器，PIC 微控制器执行每条指令需要 4 个振荡器周期。2051 每条指令的执行需要 12 个振荡器周期，甚至许多指令执行需要超过 1 条机器周期。PIC 微控制器同时还使用了流水线操作，而 Atmel 微控制器没有。

2.8 更多细节——16F84A 片上复位电路

让我们进一步看看 16F84A 片上复位电路，图 2-11 为该复位电路的简化图。虽然需要花些时间去理解，但这是值得的。

41

对 CPU 的实际复位，$\overline{\text{Chip_Reset}}$，由一个触发器产生，如图 2-11 右边所示。这个

触发器有两个输入——S（置位，Set）和 R（重置，Reset）。当 S 线为高从而致使 Chip_Reset为低时，CPU 进入复位模式。CPU 保持在复位模式直到 R 线为高而致使触发器被清零。

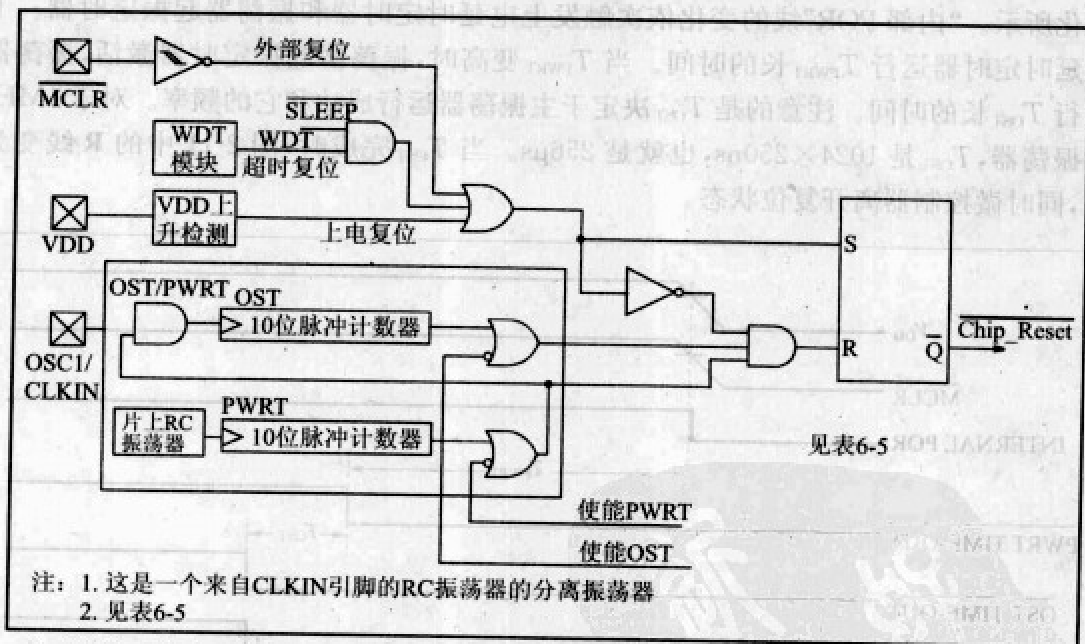
那么是什么引发复位的呢？如果下面列出的任何一种情况变为高，通过 3 输入的或门，使触发器的 S 输入变高，就会引发复位。

❑ 外部复位：来自我们已经知道的MCLR线。

❑ 超时复位：来自看门狗定时器（WDT）；超时复位被设计在程序崩溃时发生——详见第 6 章。

❑ 上电复位：检测电源上电的电路（"V_{DD}上升检测"）的输出。

一旦这 3 种情况中的任何一种发生，触发器就会被设置，Chip_Reset线变低且 PIC 微控制器维持在复位状态。



参考文献2.1的表6-5给出了不同振荡器设置的复位延迟时间的例子，以及PWRT和OST使能信号线

图 2-11 16F84A 复位电路

如果触发器的 **R** 输入被激活，Chip_Reset线回到高（且 PIC 微控制器使能，可以开始工作）。要激活触发器的 **R** 输入需要同时满足 3 个条件，这 3 个条件由与 R 输入相关的与门的输入决定，它们分别是电源和振荡器已经稳定，以及复位清零的任何条件得到满足。电源和振荡器已稳定这两个条件由两个定时器给出，这两个定时器分别是上电延时定时器（Power-Up Timer，PWRT）和振荡器起振定时器（Oscillator Start-up Timer，OST）。可通过设置配置字中上电延时定时器的配置位来使能上电延时定时器（如图 2-6 所示）。可通过 **Enable OST** 线来使能振荡器起振定时器。这个可以通过配置字中用户振荡器配置来自动设置，配置字中的用户振荡器配置可以使能除 RC 以外

42

的所有振荡器模式。上电延时定时器由它自己的片上 RC 振荡器计时，并且在上电延时定时器启动后。它的振荡器计数 1024 个节拍后设置它的输出为 1，这段时间大约为 72ms。对于通常的电源来说，这段时间对于让电源稳定足够长了，但对于上电缓慢的电源来说，这还不够。一旦上电延时定时器完成它的计数，那么振荡器起振定时器就会被激活，振荡器起振定时器对主振荡器信号按次计时 1024 个节拍。这是为了测试时钟振荡器运行是否可靠——如果时钟振荡器没有运行，那么很显然振荡器起振定时器不会计数。两个计数器的输入和触发器 **S** 输入的取反信号相与，就得到触发器的 **R** 输入。如果所有的信号都为高，也就是两个计数器都完成它们的计数并且复位的任何条件都不存在，那么触发器就被清零。相应地，CPU 离开复位状态，然后开始运行。

图 2-12 描述了$\overline{\text{MCLR}}$连接到 V_{DD}这种通常情况下的复位次序。从 V_{DD}迹线上升可看出电源上电，同时也使$\overline{\text{MCLR}}$线变为 1。这个改变被检测到，如图中"内部 POR"线的变化所示。"内部 POR"线的变化依次触发上电延时定时器和振荡器起振定时器。上电延时定时器运行 T_{PWRT}长的时间。当 T_{PWRT}变高时，振荡器起振定时器激活，振荡器运行 T_{OST}长的时间。注意的是 T_{OST}决定于主振荡器运行成功和它的频率。对于 4MHz 的振荡器，T_{OST}是 1024×250ns，也就是 256μs。当 T_{OST}完成时，图 2-11 中的 **R** 线变为高，同时微控制器离开复位状态。

43

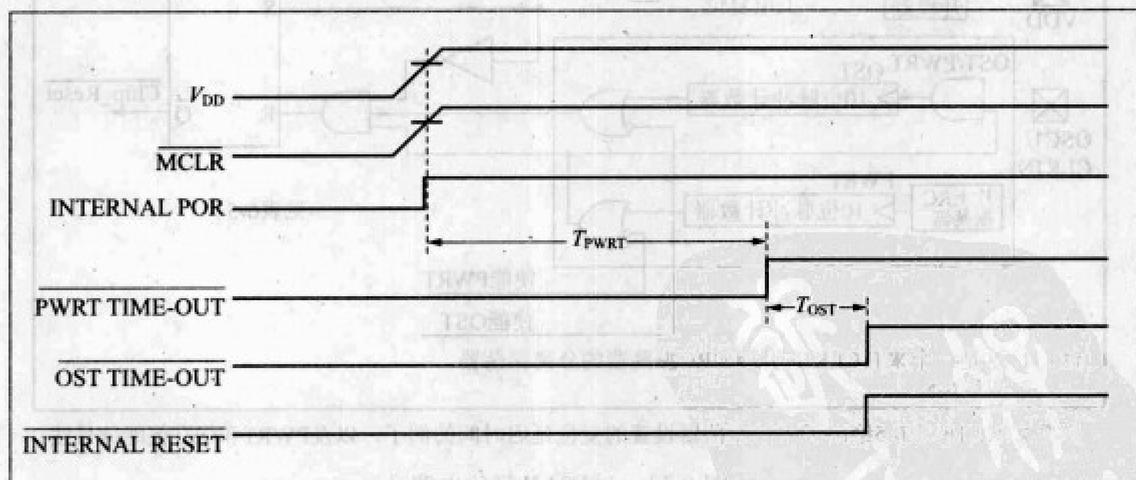


图 2-12 上电时复位时序，$\overline{\text{MCLR}}$连接到 V_{DD}

小结

- ☐ PIC 16 系列是一个种类繁多且有效的微控制器系列。
- ☐ 16F84A 采用哈佛存储器结构、流水线操作和 RISC 指令集，代表了所有 16 系列微控制器的体系结构。
- ☐ PIC 16F84A 含有有限的外围设备，这些外围设备运用于小型和低成本的应用中。因此，它只是 16 系列中一个较小的微控制器，它的所有特征都包含在较大的微控制器特征中。

- ❑ 16F84A 在其不同的存储器区域中采用了 3 种截然不同的存储器技术。
- ❑ 特殊功能寄存器是一类特殊的存储单元，它用于连接 CPU 和外围设备。
- ❑ 复位机制确保 CPU 在适当的运行条件满足时开始运行，并且可以在程序出现错误时重启 CPU。

参考文献

2.1. PIC 16F84A Data Sheet (2001). Microchip Technology Inc., DS35007B; www.microchip.com

2.2. PICmicro Mid-Range MCU Family Reference Manual (1997). Microchip Technology Inc., DS-33023 A; www.microchip.com

2.3. PIC16F84 to PIC16F84A Migration (2001). Microchip Technology Inc., DS30072B; www.microchip.com

2.4. 8-bit Microcontroller with 2K Bytes Flash (2000). Atmel Corporation, AT89C2051, Rev. 0368E-02/00; http://www.atmel.com/

2.5. Atmel 8051 Microcontrollers Hardware Manual (2004). Atmel Corporation, Ref. 4316C-8051-05/04; http://www.atmel.com/

第3章 并行端口、电源和时钟振荡器

至此，我们已经对微控制器体系结构的理论及其在PIC®微控制器中的实现有了初步了解。现在，本章开始从微控制器体系结构的理论转到小规模硬件设计实践上来。

正如我们之前所提到的，微控制器核有内部数据总线和地址总线。在某种程度上，它们就像汽车高速公路或者各州内高速公路一样，在两个方向上都有大量的交通运输，而它们的目的地各不相同。需要提供一种方式允许微控制器的数据流和外部世界连接，以便微控制器能够读入外部数值或者输出其他数值。换句话说，微控制器需要同汽车高速公路出入口一样的交叉点，在这些交叉点上数据可以在设计好的时间和位置离开（或进入）总线。在微控制器中，这些交叉点有多种形式，因为数据有多种形式的输入和输出。最通用的一种形式是并行输入/输出端口，这是微控制器最必要的外围设备之一，并且是本章一开始就要讨论的主题。

假如有一个工作的汽车引擎，它需要两个必备要素才能运行：燃料和由活塞产生的火花。微控制器有相似的需要，微控制器的燃料就是它需要的低电压电源，微控制器的火花就是规整的时钟振荡器脉冲序列。这些将在本章的后半部分学习。

把我们已经获得的背景知识联系在一起（背景知识包括能够将数字输入/输出用于设计且能够设计电源和时钟振荡器），就有能力开始设计实际的系统，我们将会为此感到非常高兴。

在本章中你将学到：

- ☐ 为什么需要并行输入/输出；
- ☐ 逻辑电路如何设计才可以在微控制器数据总线和外部世界之间提供灵活的接口（也就是并行端口）；
- ☐ 外部设备怎样才能连接到并行端口上；
- ☐ PIC 16F84A上可用的并行输入/输出；
- ☐ 电源和时钟振荡器的必要硬件特征；
- ☐ 通过16F84A介绍Microchip公司的电源和时钟振荡器方法；
- ☐ 电子乒乓球游戏的硬件设计。

3.1 并行输入/输出概述

几乎所有的嵌入式系统都需要在它的CPU和外界之间传输数字数据。这种传输可大致分为以下几类。

直接的用户接口：包括开关、键盘、发光二极管（light-emitting diode，LED）和显

示器。

输入的测量信息：来自外部传感器，可能是从模数转换器获得的。

输出的控制信息：例如电机或其他传动装置。

大量的数据传输：通过串行或并行的形式，到达或来自其他系统或子系统，例如传输串行数据到外部存储器。

存在这么多的数据往来，我们可能需要各种数字输入和输出。这些数字输入/输出大体上被分为串行和并行，在串行数据传输中，信息一次传输一位，只有一条单一的互连线用于携带数据本身，但通常还有其他线用于异步处理和控制。在并行数据传输中，使用一组（例如，8条）互连线，每条互连线携带一位数据，并且每条互连线都和其他互连线并行工作，数据能以位组方式传输，例如以字节传输。并行输入/输出（I/O）是实现微控制器的所有基本数据交互的最有效的方式，这些基本的数据交互包括与开关、LED、显示器等设备的交互。出现在微控制器引脚上的一组并行I/O互连线称为**并行端口**（parallel port）。

3.2 并行输入/输出的技术挑战

我们的直接挑战是怎样在微控制器数据和地址总线与外界之间提供所需的接口。如上所述，我们从数据总线（也就是多用途的数据公路）开始讲述。怎样才能从数据总线上获取我们想要的数据，并把这些数据传输到外界？是通过并行端口吗？怎样才能在正确的时间和位置上获取外部输入数据，并把它放在数据总线上，以便它能放在微控制器内正确的位置？最后，假定有一个端口能够完成上述任务，我们怎样才能使它真正具有灵活性，以便它能用于输入、输出、或既能输入也能输出，以及能传输可能的很多不同用途的数据组合？

3.2.1 设计并行端口

创建一组输出引脚来建立一个**输出端口**非常简单（如图3-1所示）。让我们在存储器映射中给这个端口分配一个地址。无论什么时候，只要该地址在程序中被指令选中，就会激活一条叫做**端口选择**（Port Select）的线以及一条叫做**读/写**的线。读/写线可以指出CPU是正在进行读（线为高电平）操作还是写（线为低电平）操作，这条线受端口选择线的门控[1]。数据总线的每条线都连到一个双稳态触发器，所有双稳态触发器都受端口选择线的钟控[2]。那么，在写模式下，只要端口存储位置被寻址到，数据总线上的值就锁存进双稳态触发器。双稳态触发器的输出可与外界连接。

创建一组输入引脚同样也很简单（如图3-2所示）。只需要在一只外部引脚和数据 46

---

① 端口选择线与该条线相与，控制该条线上信息的输出。——译者注

② 端口选择线通过一个与门控制读/$\overline{写}$的输出值，该输出值连接到双稳态触发器的时钟端口，控制数据锁存进双稳态触发器。——译者注

总线的一条线之间连接一个三态缓冲门。当三态缓冲门使能时,还是由端口选择线和读/写逻辑组合控制,外部引脚的逻辑值简单地与数据总线的线连接,并能被 CPU 读取。要注意的是,在这个设计中,外部数据未被端口锁存,外部数据源必须把它维持在一个稳定的值上。

图 3-1 和图 3-2 中的并行接口想法非常好,但是在实际中,由于限制 IC 的一个引脚只能有一个功能(输入或输出),所以这种方式缺乏灵活性。更好的想法是将这两个

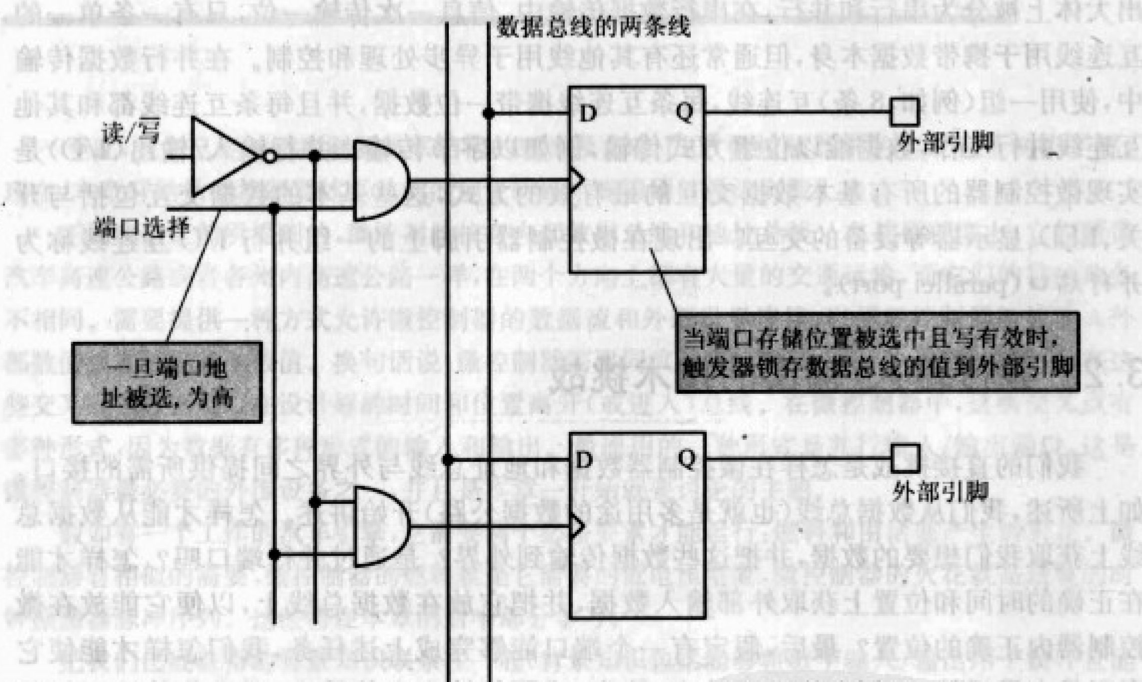


图 3-1 数字输出端口的两位

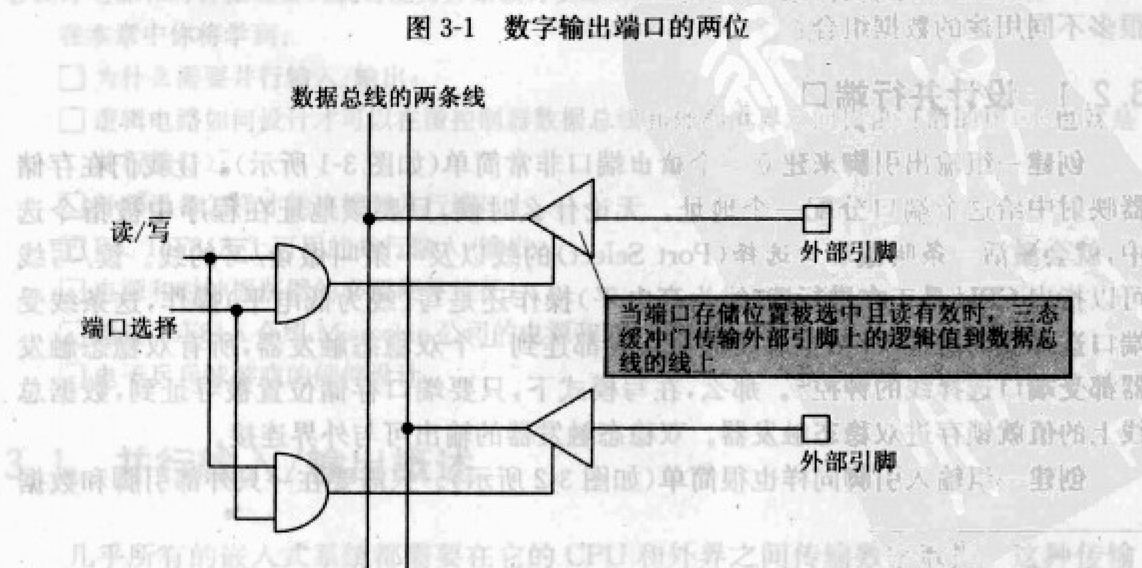


图 3-2 数字输入端口的两位

用于输入和输出的电路以某种方式结合起来，让用户决定他们想要数据移动的方向。图 3-3 就是体现这种思想的电路图，由图可以看到，并行端口每一位的“引脚驱动器”电路。从图 3-3 中可以很容易地找出图 3-1 和图 3-2 的电路。在图 3-3 中还必须加上一个触发器(“方向”触发器)，它用来确定微控制器的引脚是作为输入还是输出。“方向”触发器的状态由程序设置，它控制着“输出缓冲”，“输出缓冲”在端口处于输出模式时启用。

47~48

图 3-3 中所示的电路是非常有用的双向输入/输出引脚驱动器电路的一个基本结构，这种电路的各种版本普遍存在于众多流行的微控制器中。多条 I/O 引脚合成一组，形成一个并行 I/O 端口。然后每个“数据”触发器形成“数据”SFR(Special Function Register，特殊功能寄存器)的一位，且每个“方向”触发器形成“方向”SFR 的一位，如图 3-3 所示。每个 SFR 都被存储器映射，具有自己唯一的地址。SFR 的选择线由 SFR 的地址得到，当 SFR 被寻址到时，选择线变为高。端口选择线(Port Select)选择数据 SFR，而方向选择线(Direction Select)选择方向 SFR。

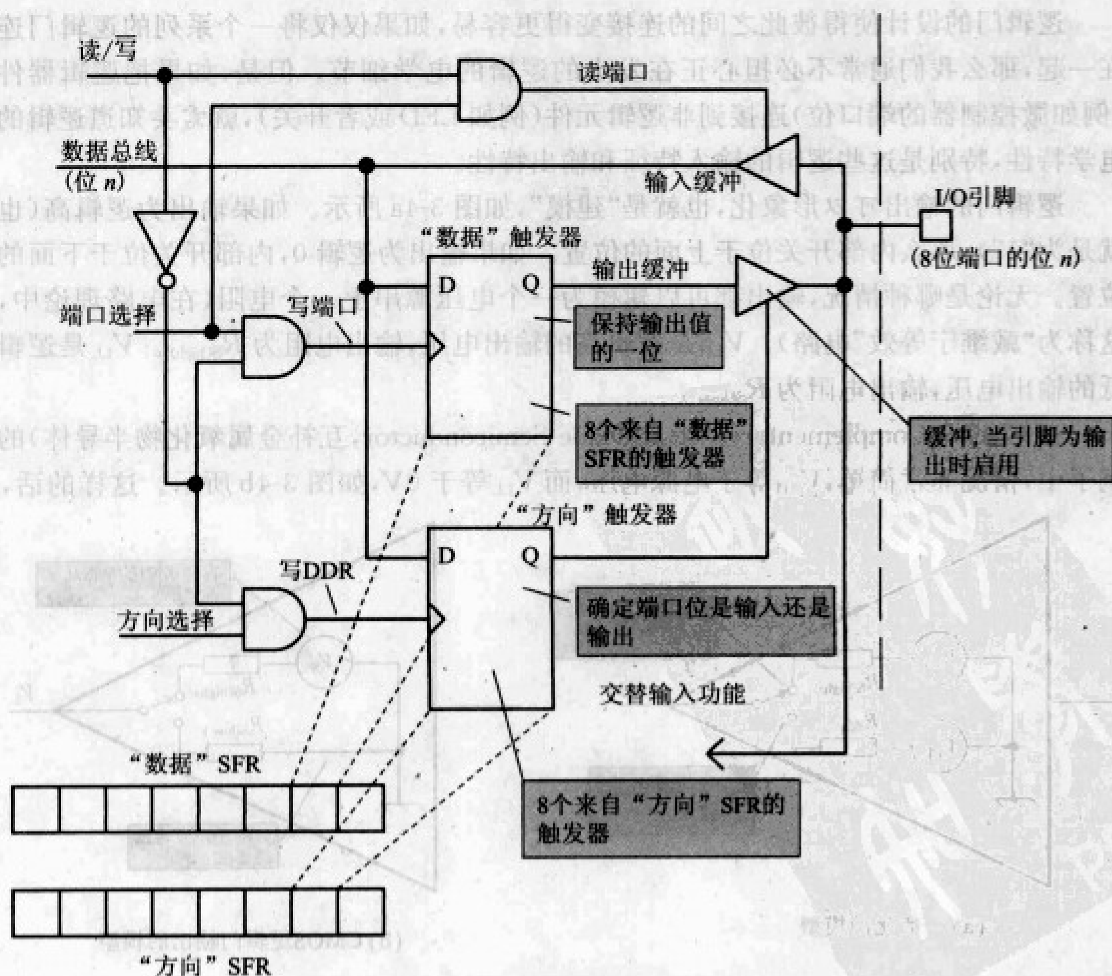


图 3-3 双向端口的引脚驱动器电路

通过写"方向"SFR,用户可以确定哪些位会被输入以及哪些位会被输出。通过写数据SFR,用户可以设置所有数据触发器的值,不管实际中引脚是被设置为输入还是输出的。引脚使能为输出时,设置在数据触发器中的值通过引脚的输出缓冲传输到I/O引脚。通过读取数据SFR,程序可以获得I/O引脚的逻辑值。如果引脚被设置为输出,读出的值仅仅是保存在数据触发器中的数据,并且它们通过输出缓冲维持在I/O引脚上。如果引脚设置成输入,那么外部信号应当连接到引脚上,并且微控制器将读出外部信号的值。

在建立这个基本设计之后,可以进一步扩展它以加入其他的特征。看一些PIC微控制器的例子时我们会见到这一点。然而,图3-3中已有一个简单的扩展特征。这个简单的扩展特征是"交替输入功能(Alternate Input Function)"线,该线允许片上外围设备共享I/O引脚。

3.2.2 端口的电学特性

逻辑门的设计使得彼此之间的连接变得更容易,如果仅仅将一个系列的逻辑门连在一起,那么我们通常不必担心正在发生的逻辑的电学细节。但是,如果把逻辑器件(例如微控制器的端口位)连接到非逻辑元件(例如LED或者开关),就需要知道逻辑的电学特性,特别是这些逻辑的输入特征和输出特性。

逻辑门的输出可以形象化,也就是"建模",如图3-4a所示。如果输出为逻辑高(也就是为"1"),那么内部开关位于上面的位置。如果输出为逻辑0,内部开关位于下面的位置。无论是哪种情况,输出都可以建模为一个电压源串上一个电阻(在电路理论中,这称为"戴维宁等效"电路)。V_{LH}是逻辑高的输出电压,输出电阻为$R_{S(high)}$。V_{LL}是逻辑低的输出电压,输出电阻为$R_{S(low)}$。

在CMOS(Complementary Metal Oxide Semiconductor,互补金属氧化物半导体)的例子中,情况非常简单,V_{LH}等于电源电压,而V_{LL}等于0V,如图3-4b所示。这样的话,

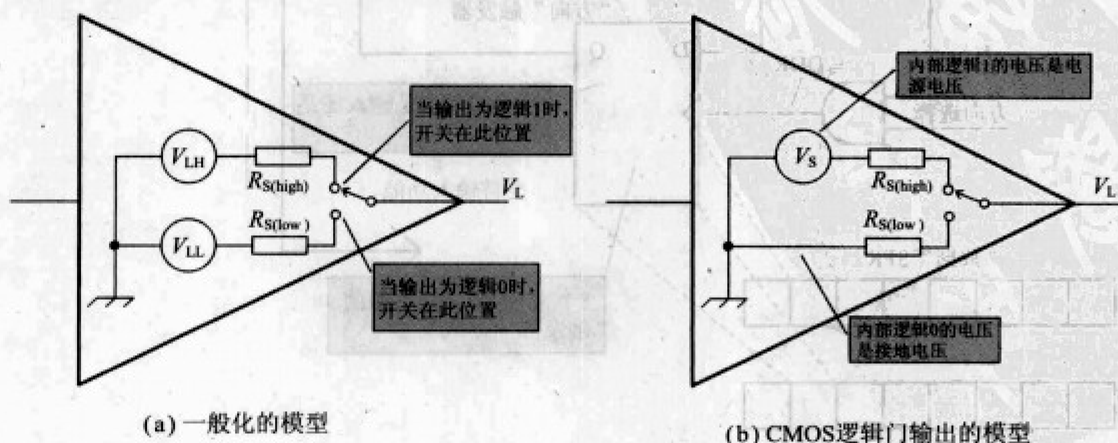


图3-4 逻辑门输出建模

如果电源电压为 5V，那么逻辑 0 和逻辑 1 将分别是 0V 和 5V（如果没有任何电流从门的输出流出）。

实际上，$R_{S(high)}$ 和 $R_{S(low)}$ 不是常数，而是在某种程度上由来自门输出的产生或消耗电流决定。所以，生产商经常发布有关输出特性的图示信息，接下来我们很快会看到 16F84A 的这些图示信息。

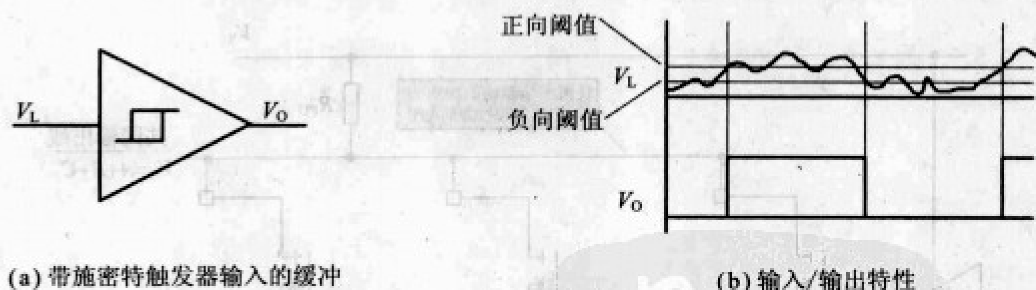
49

3.2.3 一些特殊的端口特性

我们现在学习两个特别类型的 I/O 特性，在探究 16F84A 并行端口时，这两个特性会很重要。

1. 施密特触发器输入

施密特触发器(Schmitt trigger)（如图 3-5 所示）是一个特定类型的逻辑门输入，它设计用于整形扭曲的逻辑信号。施密特触发器有 2 个输入阈值——“正向(positive-going)”阈值和“负向(negative-going)”阈值，“正向”阈值比“负向”阈值高。从一个低数值开始的信号只能经过“负向”阈值（在“负向”阈值处，什么也不发生），然后穿过“正向”阈值，这个时候，输出改变状态。输出将会直到输入已经回落到“负向”阈值时才反转。所以，小的信号波动会重新穿过刚刚穿过的阈值，但不引起输出的任何变化。



(a) 带施密特触发器输入的缓冲　　(b) 输入/输出特性

图 3-5　施密特触发器

2. “开漏”输出

开漏(Open Drain)输出是一种灵活的输出类型，它能被用作标准的逻辑输出、或直接驱动小负载，或用于叫作“线或(Wired-OR)”的特殊逻辑功能。图 3-6a 就是开漏输出。逻辑门驱动 MOSFET(Metal Oxide Semiconductor Field Effect Transistor，金属氧化物半导体场效应晶体管)的门，该 MOSFET 没有连接的漏端形成输出。当 MOSFET 的门驱动为高时，MOSFET 导通，在漏端为逻辑 0。当逻辑门为低时，MOSFET 不导通，并且（如果没有其他的连接）漏端会是一个不确定的电压。如果从漏端到电源电压连接一个上拉电阻，那么开漏输出会或多或少表现得与常规的逻辑输出相似。然而，如果没有常规逻辑输出的有效上拉，开漏输出的上升时间会比较慢，而它产生的电流量会受限于电阻的值。

50

开漏输出也可用于驱动简单的负载，如图 3-6b 所示。负载不必加上和电源电压相

同的电压，但仍须有相同的极性。因此如果所有条件都满足，那么一个电源电压为 5V（图中的 V_{S1}）的微控制器可以驱动电压为 12V（图中的 V_{S2}）的负载。

开漏输出的另一个应用是"线或"连接，如图 3-6c 所示。这里，多个开漏输出连接在一起，且都通过一个单一的上拉电阻 R_{PU} 连到高电平。如果所有的输出都关断，共同输出线（V_O）为高电平。如果任何一个输出变为低，那么共同输出线被拉到低电平。这是实现"或"或者"或非"逻辑功能的一种可能的方法，而且它对我们后面将要学习到的某种串行连接很重要。

51

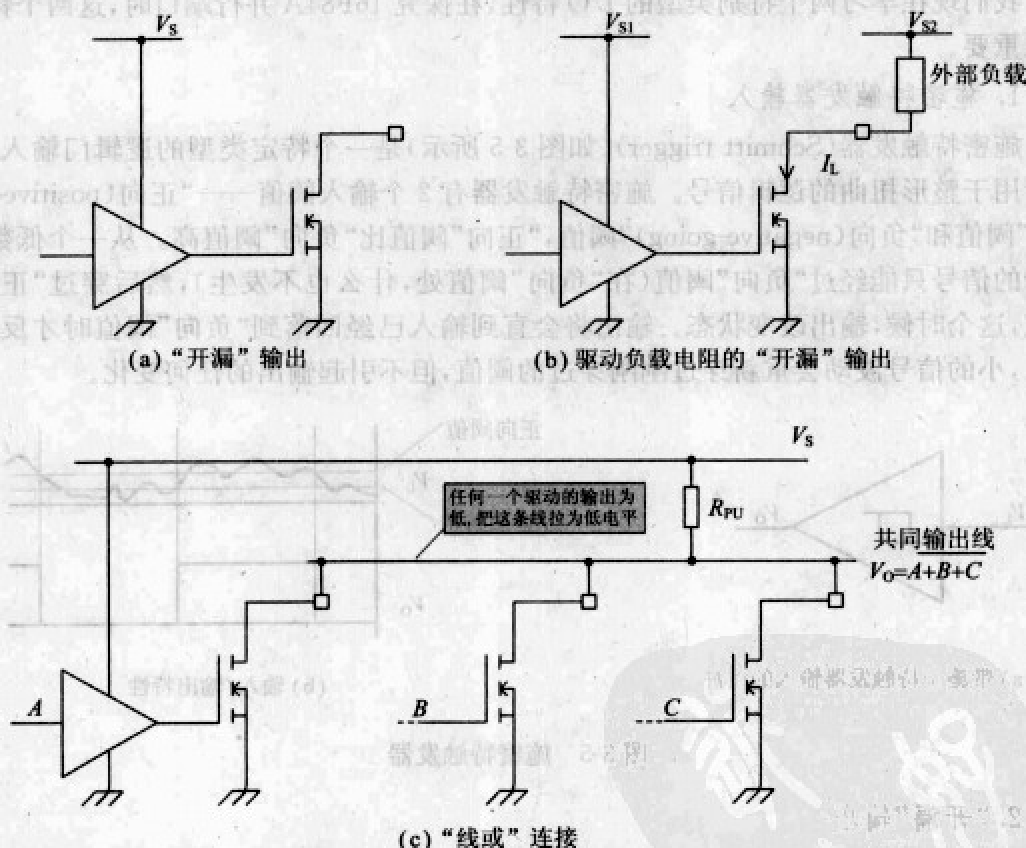


图 3-6 开漏输出和一些应用

3.3 与并行端口连接的设备

3.3.1 开关

开关广泛应用于嵌入式系统中。我们最初主要感兴趣的并不是直接开关电压或电流，而是把开关位置转换为能被微控制器端口位读取的逻辑电平。开关以按钮、拨动开关、滑动开关、指轮开关或旋转开关等形式作为直接的用户接口使用。开关还可

以以微开关的形式用于探测某种机械运动。

图 3-7a 显示了从开关中得到逻辑电平的最简单的方法。图中显示的是一个单刀双掷(single-pole, double throw, SPDT)开关，它的一端连接到地，而另一端连接到电源。该开关的接触刷只需要选择这两端中的一端作为逻辑输出。一些逻辑电路不允许逻辑输入和电源电压之间直接连接，因此需要按序串联一个电阻(如图中虚线所示)。

因为图 3-7a 的连接需要 SPDT 开关，这就使得它有一点不足。一个更简单的方法是仅使用一个单刀单掷(single-pole, single-throw, SPST)开关，例如一个按钮，如图 3-7b所示。图中显示有一个上拉电阻连接到 SPST 开关的一端，SPST 开关的另一端连接地。如果开关合上，那么逻辑门的输入 V_I 为 0V 并且有电流 V_S/R 流向地。如果开关打开，那么 V_O 等于 V_S。为了减少开关闭合时浪费的电流，R 的值应当高。然而，如果 R 值太高，按理定义的逻辑 1 电平可能不能完全维持。输入漏电流和逻辑阈值可以用来解释上拉电阻的上限值(在参考文献 1.1 的第 2 章给出了证明)。对于 PIC 微控制器来说，上拉电阻值的范围通常在 10kΩ～100kΩ 是合适的。图 3-7b 中所示的电路非常有用且被广泛使用，因为很多简单的开关(例如，安装在 PCB 上的滑动开关和按钮)的作用仅仅和 SPST 一样。

图 3-7b 中所示的开关电路可重新连接成图 3-7c 中所示的那样。然而，一些逻辑电路的特性(例如，TTL)对图 3-7c 中所示的电路的使用有限制，因为有门输入产生的电流极大地影响下拉电阻的行为。该电路可以用于 PIC 微控制器。 52

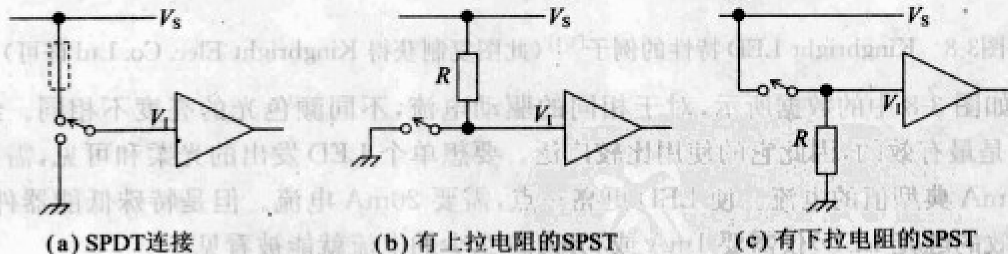


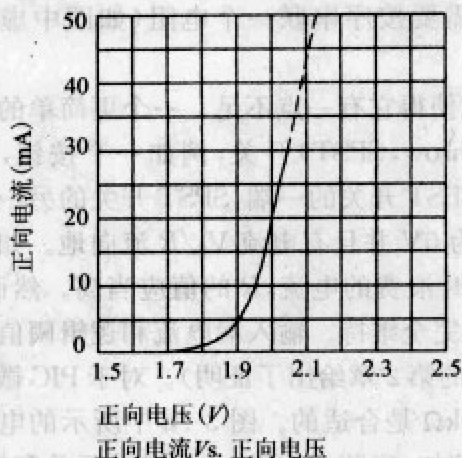
图 3-7 连接开关到逻辑输入

3.3.2 发光二极管

在某些半导体材料中，当电流流过正偏 PN 结时，会有光发出。LED 利用了这种现象。由砷化镓(gallium arsenide, GaAs)构成的 LED 发出红外区的光，如果加入磷以提高掺杂比例，发出的光线就会移动到可见红光区，最终移到可见绿光区。LED 在发出红光、绿光和黄光方面有着广泛的使用，可作为单个器件，也可作为阵列、条形图以及文字数字显示。

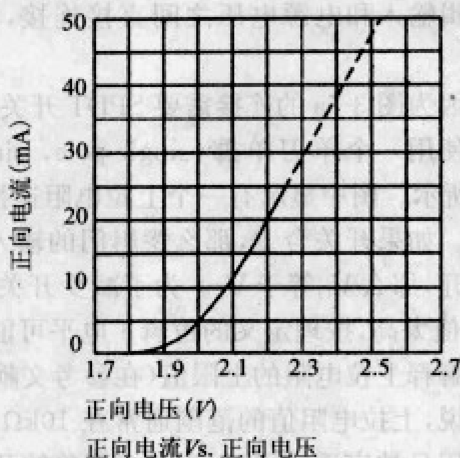
因为 LED 是二极管，LED 有正偏二极管的常规电压—电流关系。这意味着，在合理的近似下，如果 LED 导通，通过 LED 的电压是一个常数。但是，需要注意的是 GaAs 的正偏电压比硅的正偏电压大很多。图 3-8 是红光和绿光 Kingbright LED 特性的例

子。从图中可以看出，如果电流从 5mA 增加到 20mA，通过红光 LED 的电压从 1.90V 变为 2.00V。对于同样的电流变化范围，绿光 LED 电压从 1.95V 变为 2.20V。这些电压值是所有相似类型 LED 的典型值，与绿光 LED 或黄光 LED 相比，红光 LED 有稍低的正偏电压。



类型编号：L-441D
波长=627nm
15mcd typ. @ 10mA

(a) 高效的红光LED



类型编号：L-44GD
波长=565nm
12mcd typ. @ 10mA

(b) 绿光LED

图3-8 Kingbright LED 特性的例子[3.1]（此图复制获得 Kingbright Elec. Co. Ltd 许可）

53

如图 3-8 中的数据所示，对于相同的驱动电流，不同颜色光的亮度不相同。红光 LED 是最有效的，因此它的使用比较广泛。要想单个 LED 发出的光柔和可见，需要大约 10mA 典型值的电流。使 LED 更亮一点，需要 20mA 电流。但是特殊低能器件（例如高效的红光 LED）仅需要 1mA 或 2mA 这么少的电流就能被看见。

只要 LED 的电流满足要求，它可由逻辑输出（例如，微控制器端口）得到①。由于取决于端口输出能力，LED 可以连接成这样的形式——逻辑输出产生电流（如图 3-9a 所示）或者逻辑输出吸收电流（如图 3-9b 所示）。

CMOS 逻辑有对称的输出，因而它能产生和吸收几乎同等的电流，因此，图 3-9 中的每个电路都可使用。相反，TTL 逻辑产生很少的电流却能吸收相对大量的电流，因此在这种情况下，更愿使用图 3-9b 中所示的配置。

通常，有一个限流电阻和 LED 串联。考虑到电路中的电压，限流电阻的计算式如下（通常不需精确地计算）：

对于电流源：$V_{OH}=RI_D+V_D$

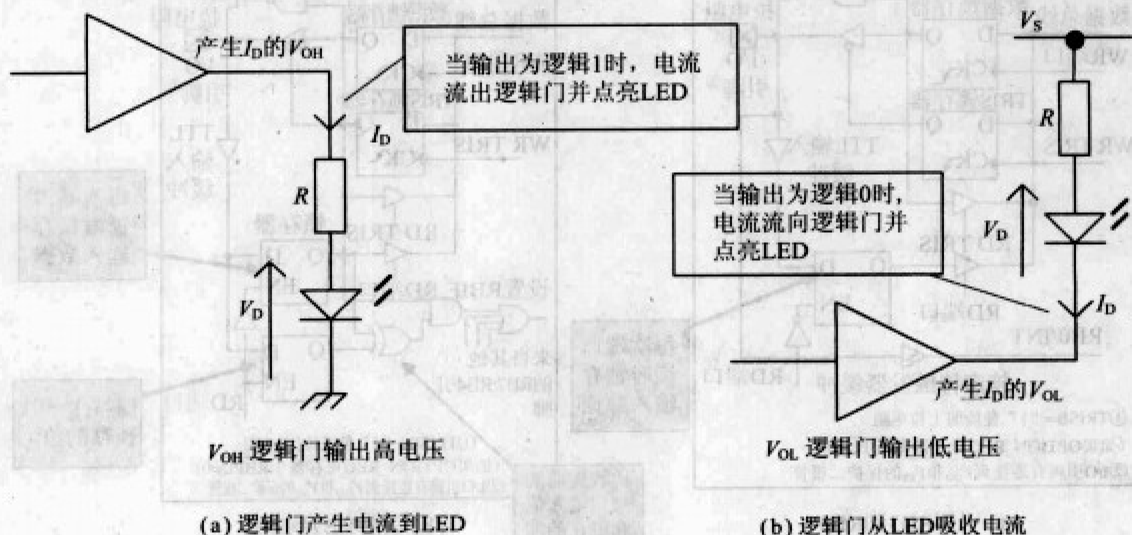
① 逻辑输出提供电流驱动 LED。——译者注

$$R=\frac{V_{OH}-V_D}{I_D} \quad (3\text{-}1)$$

对于电流阱：$V_S=V_{OL}+RI_D+V_D$

$$R=\frac{V_S-V_D-V_{OL}}{I_D} \quad (3\text{-}2)$$

采用串联电阻这种方式存在一个例外，那就是逻辑电压是由相对低的电压提供的，逻辑的内部输出电阻本身就达到了符合限流电阻的值，这种情况下就不需要再串联一个电阻了。这点在应用时需要特别注意。



(a) 逻辑门产生电流到LED (b) 逻辑门从LED吸收电流

图 3-9 从逻辑门得到 LED

3.4 PIC 16F84A 并行端口

从第 2 章中我们已经知道 16F84A 有 2 个端口：A 和 B。端口 A 为 5 位，而端口 B 为 8 位。从图 2-1 中可以看到有些端口有一个以上的功能。接下来我们将看到，16F84A 采用了图 3-3 中基本的引脚驱动器电路，并在这些优秀的功能中进行了巧妙的搭配。

从图 2-5 可以看出 SFR 与端口有关。通常，端口数据本身出现在 **PORTA** 或 **PORTB** 寄存器中(也就是，**PORTA** 和 **PORTB** 的作用和图 3-3 中的"数据"SFR 一样)，而数据的方向由 **TRISA** 或 **TRISB** 寄存器中设置的位的数值决定(即 **TRISA** 和 **TRISB** 的作用和图 3-3 中的"方向"SFR 一样)。

下面我们来探究更多关于端口的细节。端口 B 最容易理解，那么我们就从端口 B 开始讨论。

3.4.1 16F84A 端口 B

端口 B 是一个通用 8 位双向端口，有和图 3-3 中电路相似的引脚驱动器电路。图 3-10a显示了端口 B 最基本的位，位 0 到位 3。在每个端口电路中都有数据锁存器，而图 3-10 中的“TRIS 锁存器”代替了之前图 3-3 中的“方向”触发器。从图中可以看出，如果“TRIS 锁存器”的输出设置为 0，那么它驱动的缓冲被使能，同时端口位处于输出模式。

(a) 引脚RB3到RB0

(b) 引脚RB7到RB4

图 3-10 端口 B 引脚驱动器电路框图

（阴影框中所附标签为作者所加）

端口 B 的电路与之前所见的简单的引脚驱动器电路相比增加了以下 4 个方面的内容。

- 通过图中位置最低的锁存器，锁存输入数据，而不仅仅是读取输入数据的瞬时值。
- 通过由 **RD TRIS** 线控制的缓冲可以读取“**TRIS** 锁存器”的状态。**TRIS** 锁存器的作用和通常的读/写存储单元一样，而且如果有必要，程序可以检查先前保存在 **TRIS** 中的数据值。
- 位 0 也是外部中断输入，并且有一个施密特触发器接口。
- 对于所有用作输入的端口位，“弱上拉”电阻[①]接通。该电阻用于替换类似图3-7b 电路中电阻。这个上拉电阻由一个 P 沟道 MOSFET 实现，见图 3-7b顶部。它们可以通过清除 **OPTION** 寄存器中的$\overline{\text{RBPU}}$位，从而使能所有被设置为输入的端口位。（**OPTION** 寄存器的存储器映射见在图 2-5，**OPTION** 寄存器全面的介

① 当端口位设置为输入时，MOSFET 的栅极输入为低电平，整个 MOSFET 导通，相当于一个电阻，这个电阻值较小，上拉电平的作用不强，所以叫弱上拉电阻。——译者注

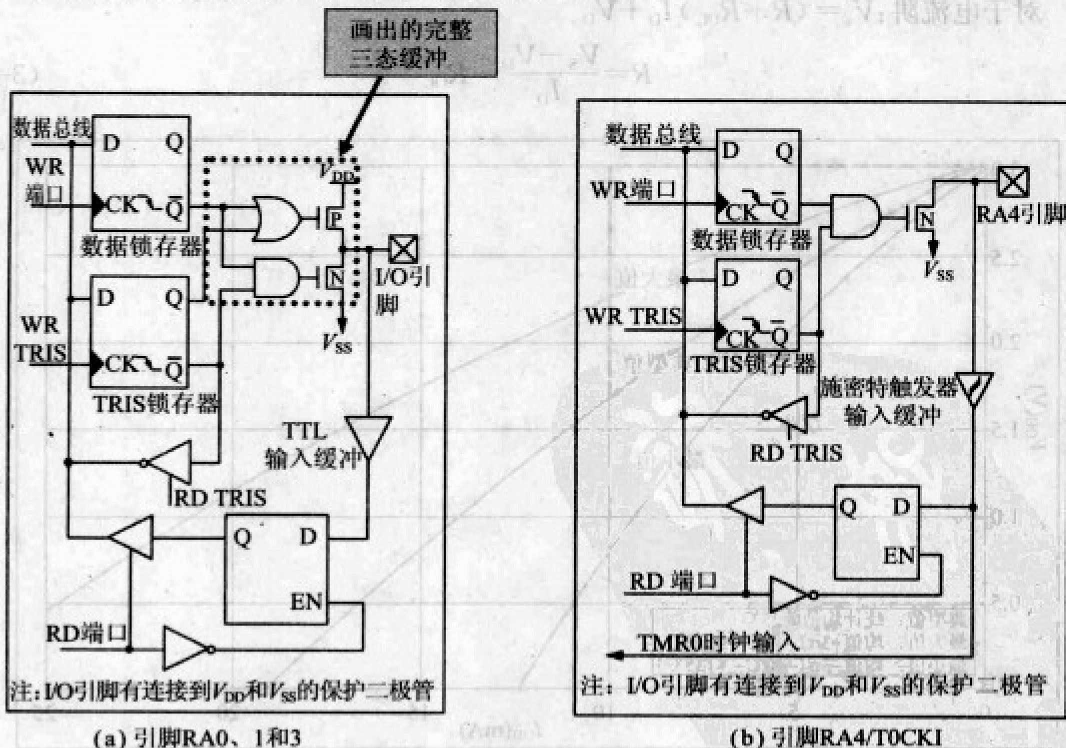
绍见图 6-9)。

从如图 3-10b 中可见端口 B 的位 4 到位 7。它们具有"电平变化产生中断(Interrupt-on-change)"的功能。与较低编号的端口位一样,数据值在读取输入数据时被锁存。然而,在这些位上,上一次端口被读的数据保存在另一个锁存器中。它保存的值和现在的输入数值有任何不同都会被电路中的异或门检测到,异或门的输出可以产生一个中断。这点将在第 6 章详细讨论。

3.4.2 16F84A 端口 A

55

和端口 B 一样,端口 A 可以作为一个通用双向数字端口。端口 A 的基本引脚驱动器(如图 3-11a 所示)和端口 B 的引脚非常相似。图 3-11a 完整地画出了输出三态缓冲。端口 A 第 4 位的功能复用,用作 Timer 0 时钟输入(如图 3-11b 所示)。位 4 也有施密特触发器输入特性和开漏输出,这两点在 3.2.3 节讨论过。完整的器件数据指出,加在开漏引脚上的绝对最大允许电压为 8.5V。因此,驱动比微控制器本身的电源电压更高的外部负载的能力只能应用于有限的地方。



(a) 引脚RA0、1和3　　(b) 引脚RA4/T0CKI

图 3-11 端口 A 引脚驱动器电路框图

(阴影框中所附标签为作者所加)

3.4.3 端口的输出特性

图 3-12 为 16F84A 端口输出特性,电源电压为 3.0V。在图 3-12a 中,我们可以看

到，当输出电流为 0 时输出电压为 3V，此时代表逻辑 1，但是当输出电流为 10mA 时（输出电流流出逻辑门），输出电压下降到 1.7V 左右（温度为 25℃时）。同样地，在图 3-12b中，我们可看到，当输出电流为 0 时输出电压为 0V，此时代表逻辑 0，但是当输出电流为 22.5mA 时（输出电流流入逻辑门），输出电压上升到 0.8V 左右（温度为 25℃时）。只要知道 I_D的值，就可以使用这两条曲线由式(3-1)和式(3-2)算出 V_{OL}和 V_{OH}。在 16F84A 的完整数据手册中也给出了电源电压为 5V 的特性曲线图。

56

使用这两条曲线还可以推导出输出电阻的近似值。可以通过测量在某一点处曲线的梯度大小来得到输出电阻的近似值。在每个图中加入的虚线用于求输出电阻的值。用图中垂直的数值（电压）除以水平的数值（电流）可推导出电阻，在逻辑高电平时输出电阻约为 130Ω，在逻辑低电平时输出电阻约为 36Ω。我们把这两个值分别称为 R_{OH}和 R_{OL}，那么式(3-1)和式(3-2)可以写成另一种不同的形式：

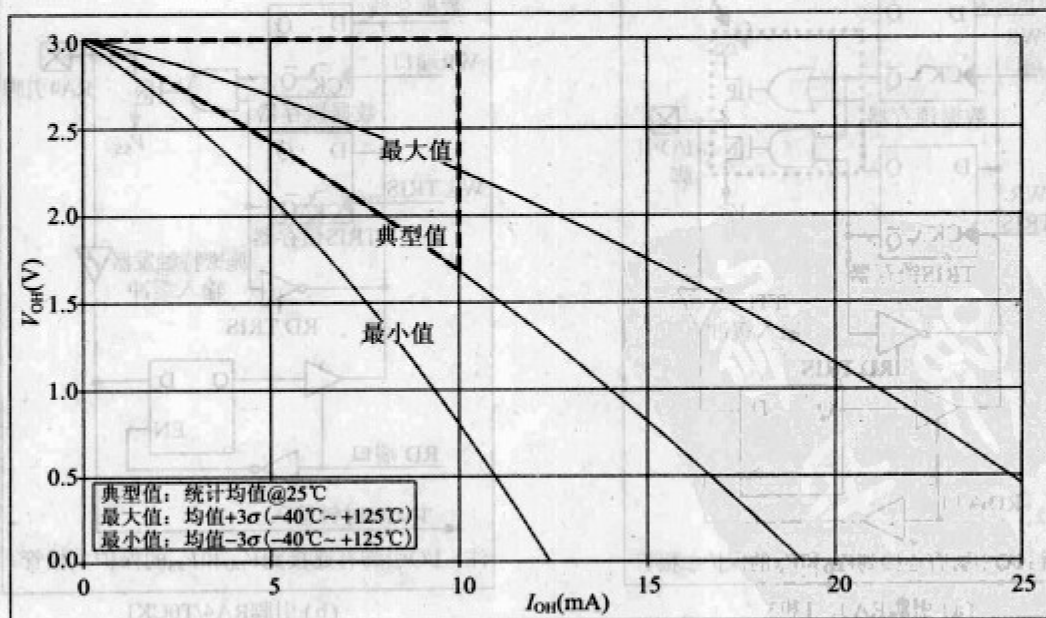
对于电流源：$V_s=(R+R_{OH})I_D+V_D$

$$R=\frac{V_S-V_D}{I_D}-R_{OH} \tag{3-3}$$

对于电流阱：$V_s=(R+R_{OL})I_D+V_D$

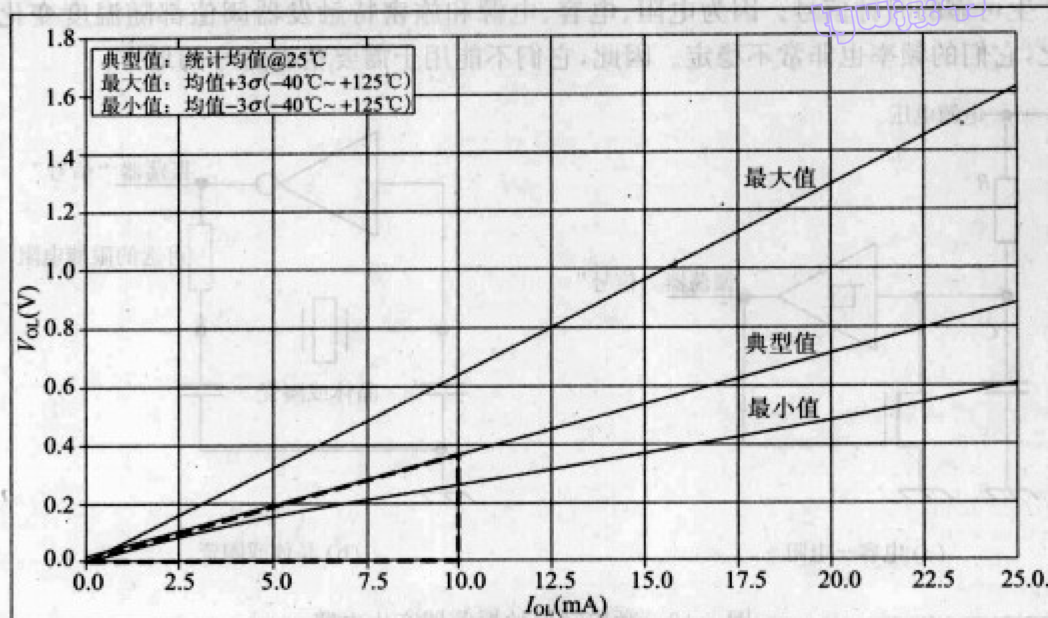
57

$$R=\frac{V_S-V_D}{I_D}-R_{OL} \tag{3-4}$$



(a) V_{OH} vs. I_{OH}(V_{DD}=3V, −40~125℃)

图 3-12　16F84A 端口输出特性（框中所附标签为作者所加）



(b) V_{OL} vs. I_{OL}(V_{DD}=3V，−40℃~+125℃)

图 3-12 （续）

3.5 时钟振荡器

微控制器时钟源的选择决定了微控制器的一些基本的运行特性。就运行速度和编程执行来说，时钟“越快越好”，而就功耗甚至可能的电磁干扰而言，时钟“越快越不好”。微控制器中所有时控单元都几乎决定于时钟特性。要想时序稳定且精确，那么时钟振荡器必须稳定且精确。上述内容应该牢记，选择时钟源时应该了解这些并谨慎选择。本节先了解可用的时钟技术，然后再学习 16F84A 所提供的时钟振荡器。

3.5.1 时钟振荡器类型

广泛地讲，在微控制器中通常使用两种类型的振荡器电路，如图 3-13 所示。在电阻—电容(resistor-capacitor，RC)类型中(如图 3-13a 所示)，电容通过电阻从电源处充电。电容电压驱动施密特触发器的输入。当电容电压超过施密特触发器阈值时，施密特触发器的输出变为高，导通与输出相连的 MOSFET，电容迅速放电，施密特触发器输出变为低，MOSFET 被关断，然后充电过程又开始。只要电源维持供电，该过程将会持续下去。时钟信号来自于在施密特触发器输入端产生的矩形波形。图 3-13a 中所示的这种简单的电路集成在许多较大的需要时钟信号的 IC 中。在这种情况下，用户通常需要在芯片外部连接电阻和电容，选择不同的电阻和电容来设置想要的频率。但是，需要注意的一点是 RC 振荡器完全可以在片上实现。RC 振荡器的成本非常低并且可

以产生可靠的时钟信号。因为电阻、电容、电源和施密特触发器阈值都随温度变化而变化，它们的频率也非常不稳定。因此，它们不能用于需要精确时序的地方。

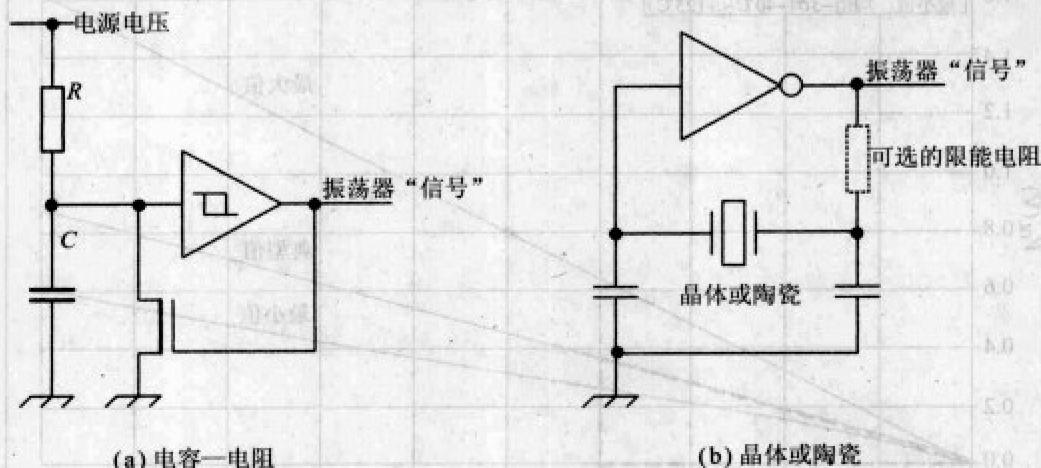


图 3-13　微控制器的振荡器产生电路

58～59

晶体振荡器(如图 3-13b 所示)依赖于石英晶体的压电属性。材料的任何机械扭曲都会产生穿过扭曲反方向的电压。同样地，如果在材料上加电压，会导致机械扭曲。小心地将晶体切成非常薄的切片(通常是圆形)，晶体会有微小的电极，将其固定好让它们可以振动。当晶体连接到含有一个逻辑反相器的负反馈通路时，如图中所示，通过压电效应强制晶体进入机械振动。机械振动变为电子振动，电子振动由逻辑门的行为维持。连接晶体电容的一边与地的小数值电容优化该振荡器所需的电子条件。

晶体以固定的且相当稳定的频率振动——这是晶体振荡器的一个很大的优势，但晶体价格昂贵(虽然成本逐渐降低)且易碎。另一种选择是陶瓷谐振器，陶瓷谐振器有和晶体相似的压电属性，并且以同样的方式连接。然而，陶瓷谐振器的成本较低但在频率上不太稳定。当需要由时钟振荡器得到精确时序功能时，晶体是唯一的选择。

3.5.2　实际使用振荡器时要考虑的问题

所有的微控制器生产商为了能够很容易地给他们的微控制器产生时钟波形，做了大量的工作。要做到给微控制器产生时钟波形通常需要在片上包含图 3-13 中所示的电路或电路的合并形式。所以，如果有人认为在微控制器上安装振荡器是一件很容易的事情，但这实际上非常复杂。不可靠的或者不能工作的振荡器都会使电路设计初学者遇到麻烦。振荡器的频率或多或少决定于电源电压、温度、湿度、PCB 板以及可能的其他因素。晶体对 PCB 板特别敏感，要求非常高。采用非常短的 PCB 迹线来排除寄生电阻、寄生电容或寄生电导很重要，因此，晶体放置在靠近微控制器的位置。

3.5.3　16F84A 的时钟振荡器

16F84A 可工作在 4 种不同的振荡器模式下，这 4 种模式可由 RC 振荡器、晶体振

荡器或陶瓷振荡器来实现，它们的详细信息如下。16F84A 还可以接受外部时钟源，用户通过设置配置字中的位来选择使用那种振荡器模式（如图 2-6 所示）。

- XT——晶体。这是标准的晶体配置。晶体或谐振器期望的频率范围为 1MHz ~4MHz。
- HS——高速。这是 XT 配置的一个有较高驱动电流的版本。较高频率的晶体以及通常的陶瓷谐振器需要较高的驱动电流。晶体和谐振器期望的频率在 4MHz 或更高的频率区域。HS 配置在所有振荡器模式中有最高的电流消耗。
- LP——低功耗。该模式供低频晶体使用并且消耗的能量最低。在大多数情况下，LP 的频率会有 32.768kHz（也就是 2^{15} Hz），这个频率广泛用于低功耗和对程序执行时间敏感的应用，例如手表。然而，LP 模式不只工作在 32.768kHz，它可以工作在约 200kHz 以下的任何频率。
- RC——电阻-电容。该模式必须有一个外部电阻和电容连接到引脚 16，采用图 3-13a 中的电路。这是得到振荡器的最廉价的方法，但是在需要精确的时序时不应使用 RC 模式。该模式的频率仅能做有限精度预测，即使这样，预测的频率也会随着温度、电源电压和时间的改变而改变。一个使用 RC 振荡器的例子是在本章末尾将要学习的电子乒乓球游戏。

60

如图 2-1 所示，16F84A 有 2 个振荡器引脚，OSC1（引脚 16）和 OSC2（引脚 15）。在这两个引脚之间有一个逻辑反相器和相关的电路。图 3-14 显示的是采用这两个引脚连接的振荡器配置。连接晶体或者陶瓷产生图 3-14a 中的振荡器电路。图中大概描述的三种不同频率范围的振荡器都可通过配置字来调用，RC 振荡器也可使用，如图 3-14b所示。振荡器频率的近似值可从 16F84A 数据手册的 Electrical Characteristics 一节所提供的图表信息中得到，例如图 3-15 所示。最后，一个外部时钟源可以简单地接到 OSC1 引脚（图 3-14c）。从参考文献 3.2 中可以获得有关 Microchip 公司微控制器振荡器设计的更深入的指导。

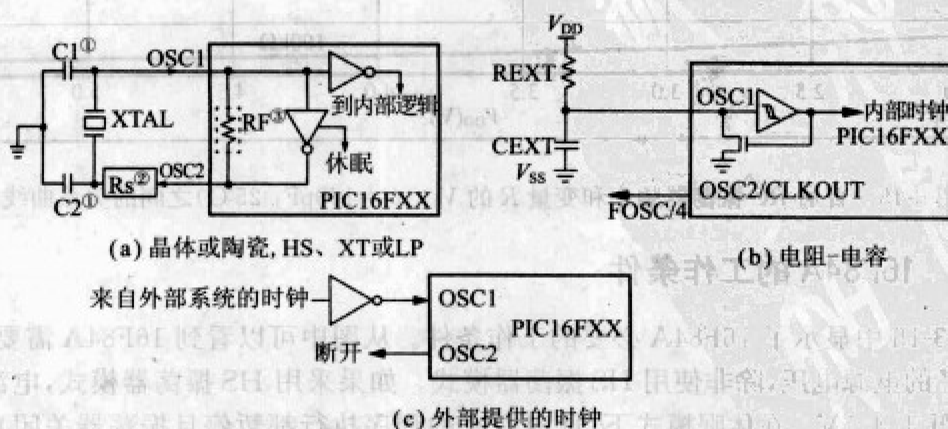


图 3-14 给 16F84A 提供时钟波形的方式

3.6 电源

3.6.1 对电源的要求

和所有电子电路一样，微控制器以及整个嵌入式系统也需要电能。传统上，多数逻辑电路的供电为 5V，这是来自于 TTL（Transistor Transistor Logic，晶体管-晶体管逻辑）逻辑电路指定的电压。随着电池供电设备的增加和电子技术的发展，电源电压已经降低，现在常用的电压为 3.3V 和 3.0V。

61

生产商的数据手册中指出了电子元件的工作条件。对于电源电压而言，有两个重要的问题：需要的电源电压以及器件能从该电源电压得到的电流。这个电流取决于工作频率。生产商的数据手册中也给出了**绝对最大额定值**（absolute maximum rating），绝对最大额定值规定了器件禁止超过的电压和电能消耗量[①]。

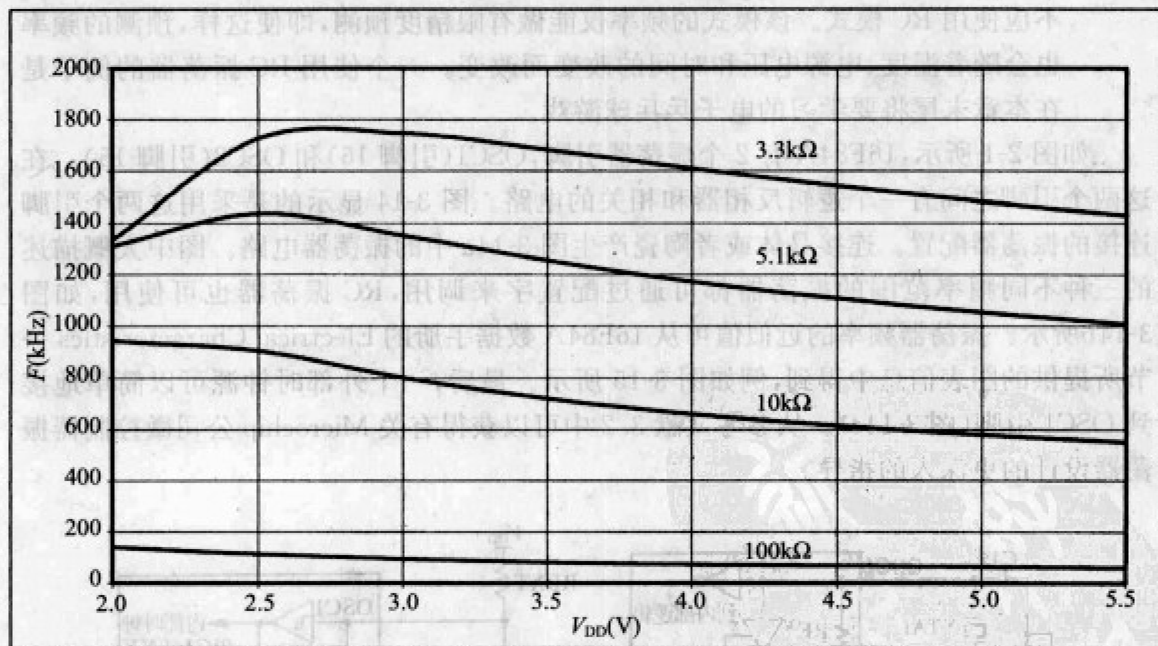


图 3-15　普通 RC 振荡器频率和变量 R 的 V_{DD}（C=100pF，25℃）之间的关系曲线

3.6.2 16F84A 的工作条件

图 3-16 中显示了 16F84A 必要的工作条件。从图中可以看到 16F84A 需要 4.0V 到 5.5V 的电源电压，除非使用 HS 振荡器模式。如果采用 HS 振荡器模式，电源电压也不得低于 4.5V。在休眠模式下（此时所有的程序执行都暂停且振荡器关闭），电源

---

① 即：绝对最大额定值定义了器件的最恶劣工作条件。——译者注

电压可以直接降到 1.5V 并且 RAM 中的数据仍然可以保留。如果需要在较低的电源电压下工作,那么应当使用 16LF84A。

参数编号	符号	特性	最小值	典型值	最大值	单位	条件
	V_{DD}	电源电压					
D001		16LF84A	2.0	–	5.5	V	XT、RC和LP振荡器配置
D001		16F84A	4.0	–	5.5	V	XT、RC和LP振荡器配置
D001A			4.5	–	5.5	V	HS振荡器配置
D002	V_{DR}	RAM数据保存电压①	1.5	–	–	V	器件处在休眠模式
D003	V_{POR}	确保内部上电复位信号的V_{DD}启动电压	–	V_{ss}	–	V	详见有关上电复位的章节
D004	S_{VDD}	确保内部上电复位信号的V_{DD}上升率	0.05	–	–	V/ms	
	I_{DD}	电源电流②					
D010		16LF84A	–	1	4	mA	RC和XT振荡器配置③ F_{OSC}=2.0MHz, V_{DD}=5.5V
D010		16F84A	–	1.8	4.5	mA	RC和XT振荡器配置④ F_{OSC}=4.0MHz, V_{DD}=5.5V
D010A			–	3	10	mA	RC和XT振荡器配置 F_{OSC}=4.0MHz, V_{DD}=5.5V (在FLASH编程时)
D013			–	10	20	mA	HS振荡器配置(PIC16F84A-20) F_{OSC}=20MHz, V_{DD}=5.5V
D014		16LF84A	–	15	45	μA	LP振荡器配置 F_{OSC}=32kHz, V_{DD}=2.0V WDT失效

① 这是在SLEEP模式下不丢失RAM数据的V_{DD}电压下限

② 给出了影响电源电流因数的更多信息

③ 指导使用外部RC网络如何计算消耗的电流

④ 表中的RC振荡器配置,未含通过R_{EXT}的电流。R_{EXT}的电流可通过公式估算出:
$I_R = V_{DD}/2R_{EXT}$ (单位:mA),且R_{EXT}单位为kΩ

图 3-16　16F84A 的基本工作条件

继续往下看表格,我们可以看到电源提供的电流是多么依赖于振荡器频率。当器件运行在 5.5V 电压和 4MHz 下时,可得到一个典型的电源电流值 1.8mA。如果振荡器的频率增加到 20MHz,那么电源电流上升到 10mA。需要注意的是,与众多其他需要更多能量的微控制器器件相比,这两个数值实际上是非常好的。但是,如果我们想在真正的低电流下工作,那么看看 16LF84A 在低频时能提供多大的电流——令人惊愕的 15μA。

你可能看出,对于用电池供电的系统,16F84A 的电源电压需要 3 个碱性电池才能有效。3 个碱性电池提供大约 4.5V 的电源。假如你采用 3 个 AA 电池供电,每个电池的电量为 800mAh。运行在 1.8mA,电池可使用 444 小时,也就是 18.5 天。运行在 10mA,电池可使用 80 小时,也就是 3.3 天。而运行在 15μA,电池可使用 53 333 小时,也就是 2 222 天,也就是超过 6 年!在这种情况下,电池自放电消耗的电量可能会占很大一部分。上面的计算当然仅仅考虑微控制器的电能消耗,而没有考虑电路的其他部分。

62

通过“休眠”模式是节能的一个重要方法。这点会在6.6节中介绍。

3.7　电子乒乓球游戏的硬件设计

在第1章中介绍了电子乒乓球游戏。电子乒乓球游戏的电路图见附录2中图A2-1。现在我们来学习电子乒乓球电路设计的细节。电源由两个AAA电池提供，这两个电池通过一个开关连接到微控制器的V_{SS}和V_{DD}引脚。因为电源电压只有3V，要使用微控制器的LF版本。电源对面的100nF去藕电容消除由于微控制器内部电路动作引起的电压毛刺。$\overline{\mathbf{MCLR}}$只是简单地连在电源线上，因为在这个简单的游戏中不需要复位功能。

63

从图中可以看到，电路中使用了一个RC振荡器。这一点非常合理，因为这个应用对成本很敏感，而电子乒乓球游戏中也没有对时序要求严格的元件。电子乒乓球游戏电路中RC振荡器的电阻为10kΩ，图3-15中显示了这个值的曲线。在电源电压为3.0V时，振荡器的频率为800kHz。

现在让我们来看一看并行端口。可以看到两个选手“球拍”连接到端口A的第3位和第4位，并有10kΩ的上拉电阻(采用图3-7b的电路样式)。“分数”LED和“出局”LED占用了端口A的剩余位，而所有“球飞行”LED与端口B相连。所有LED都是高效类型，并都按照图3-9a的方式连接。在本章3.4.3节算出的输出电阻为130Ω，那么和每个LED串联的总电阻为(560+130)Ω。通过LED的正偏电压大约1.8V，由此通过等式(3-3)可得到电流，即：

$$I=(3-1.8)/(560+13)$$
$$\approx 1.7\text{mA}$$

这个电流值对电子乒乓球这类应用和LED来说勉强合适，只需要仔细的观察就可以得出。采用这种方式得到的值通常是不精确的。

小结

- ☐并行端口允许外部世界和微控制器CPU之间数字数据的快速交换。
- ☐了解并行端口的电子特性以及它们如何与外部元件交互非常重要。
- ☐虽然有大量不同的端口逻辑设计，但这些不同的设计趋于采用相似的电路样式。端口的内部电路值得了解，因为它能使端口有效使用。
- ☐16F84A有各种不同且灵活的并行端口。
- ☐微控制器能工作的必要条件是要有时钟信号。时钟振荡器的特性决定了工作的速度和时序的稳定性，同时极大地影响微控制器的功耗。振荡器中起主要作用的元件通常内置于微控制器中，但是设计者必须选择振荡器的类型、频率和配置。
- ☐微控制器要能工作还需要电源。设计者要知道需要怎样的电源，并必须在设计中提供合适的电源。

参考文献

3.1. Kingbright Elec. Co. Ltd; http://www.kingbright.com.tw

3.2. Overview, Design Tips and Troubleshooting of the PICmicro™ Microcontroller Oscillator (2001). Microchip Technology Inc., DS33023A; www.microchip.com

64

第4章 编程伊始——汇编介绍

嵌入式系统设计包括2个主要方面——硬件和软件。在微处理器早期，采用大量集成电路(Integrated Circuit，IC)建立系统是非常费力的。存储器容量非常有限，所以程序员只能编写小程序。逐渐地，可用的IC变得越来越复杂，而程序员也可以编写较长的程序了。当前，存储器品种繁多且价格便宜，同时硬件复杂却易于使用。通过相当轻松的工作就能建立复杂的硬件系统，在许多项目中，软件开发是主要的创造性活动。在本章中，我们将开始编写程序，这将是一段漫长而又令人兴奋的旅程。我们首先使用汇编语言来编写程序，而在本书的后面将使用高级语言C语言来编写。

一旦开始编程，我们就会遇到一个问题：程序将在什么上运行？当然，编写的嵌入式系统程序最终是在目标硬件系统上运行的。你可以在一个教学PIC® 硬件系统上运行编写的程序，也可以在电子乒乓球游戏硬件系统上运行。然而，在许多情况下，我们并不想在实现编程思想时由于硬件环境的因素而受到限制。使用仿真器能非常方便地促进编程学习。仿真器是一个运行在台式计算机上的程序，它能够运行已经开发的嵌入式系统程序。因此，本章首先介绍Microchip MPLAB® 集成开发环境，以及该开放环境中的仿真器。一旦学会使用它，就可以非常快速地编写和运行程序来验证编程思想，同时在优秀且复杂的微控制器编程技术上，你将能获得快速的进步。

在本章中将学到：

- ❑ 计算机编程的一些基本问题；
- ❑ 汇编语言编程的要素以及怎样编写简单的汇编程序；
- ❑ 用于编程的开发环境和Microchip MPLAB® 集成开发环境；
- ❑ PIC 16系列指令集回顾；
- ❑ 某些PIC 16系列指令的使用；
- ❑ 仿真软件和MPLAB软件仿真器MPSIM™。

如果你愿意，还将学到：

- ❑ PIC 16系列的RISC指令集与功能相当的CISC微控制器的指令集之间的比较；
- ❑ PIC 16系列指令字的详细结构。

4.1 程序功能与开发流程

计算机编程包含以下4个主要思想。

(1) 计算机含有指令集;它能识别并执行每条指令。

(2) 计算机执行的程序是从它的指令集中抽取的一列指令;计算机以二进制的形式从它的程序存储器中读出这些程序。这种形式的程序称为**机器码**(machine code)。

(3) 程序执行时,计算机从程序起始处完整地按照程序指令来工作,严格地做每条指令要求做的事情,不会多做,也不会少做,除了在中断引起程序转向时。

上述3点计算机编程思想较简单,但下面这点较复杂:

(4) 程序员必须有方法能将他的思想分解并转换成计算机能够接受的各个步骤,这里的每个步骤最终必须是来自计算机指令集的一条指令。

4.1.1 编程问题和汇编折中方案

图4-1概括了编程的问题。通常,我们以复杂的并且通常是模糊定义的语言形式来表述我们的思想。计算机读取并"理解"二进制,并且对精确的指令做出精确的反应。计算机是逻辑结构,毫无情感,只会严格地做它被告知的事情。

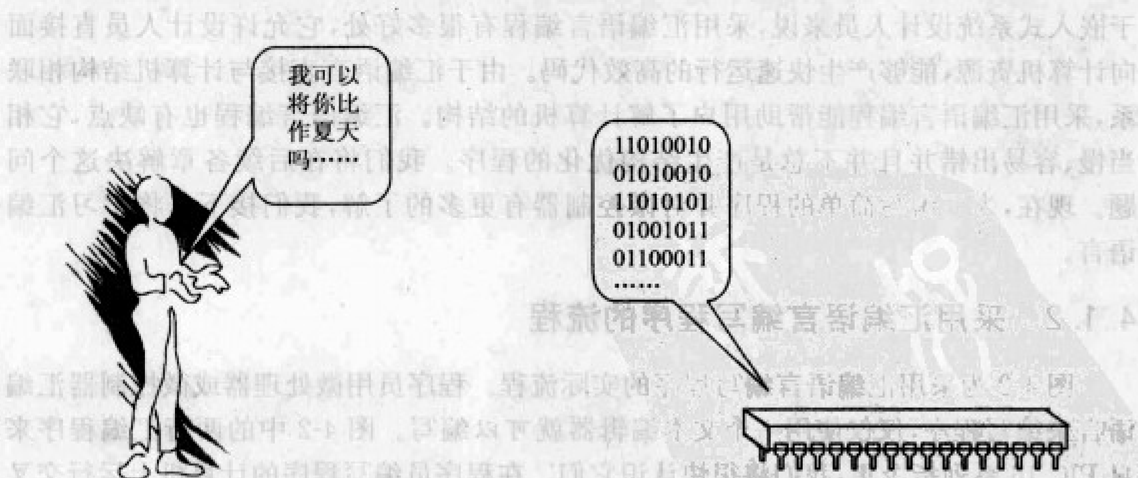


图4-1 编程问题

假使在这种语言差异下,程序员怎样编写计算机使用的程序呢?有3种方式来弥补这种差异,如下所示。

(1) **人类学习机器码**。在早期编程中,有时候程序员习惯于这样做。正如计算机以二进制代码的形式来读取指令一样,程序员花费很大的力气以计算机二进制代码的形式来编写每条指令。这项工作非常缓慢、极度无聊并且容易出错,但是至少程序员能直接面对计算机的需求和处理能力。

(2) 使用高级语言(High-Level Language,HLL)。这就像我们要求计算机学习我们的语言一样。在 HLL 中,我们用自己所能识别的语言来编写指令——对于阅读本书的人来说,所使用的语言为英语,也就是用英语来编写指令。之后,另一个计算机程序(编译器或者解释器)将编写的程序转换为计算机能够理解的机器码。程序员现在就会很轻松而且能编写非常复杂的程序了。然而,由于程序员不能接触到计算机资源,所编写的程序在存储器使用和执行速度上效率会相对较低。

(3) 使用汇编(Use Assembler)。这是解决编程问题的一个折中的方案。计算机指令集的每条指令都有一个助记符(mnemonic)。助记符通常是 3 个或 4 个字母的单词,它用于直接表示指令集的一条指令。那么程序员就可以使用指令助记符来编写程序了。由于程序员直接面向计算机指令工作,他们必须在计算机层面上进行思考,同时他们使用了助记符,实际上就不必针对计算机机器码来工作了。一个特殊的程序将采用助记符编写的代码转换为计算机能识别的机器码,这个特殊程序称为交叉汇编器(Cross-Assembler),目前通常在 PC 机上运行。计算机可以将汇编代码转换为机器代码,这给编程带来很大的好处。例如,交叉汇编器能够处理与程序存储器存储空间分配相关的大多数问题,它能够识别用于表示数字和存储单元的标号,这极大地减轻了程序员的工作。

在早期计算领域,汇编语言几乎用于为所有类型的计算机编写程序。然而,如今仅嵌入式系统设计人员仍在大量使用汇编语言,特别是在使用较小的 8 位器件时。对于嵌入式系统设计人员来说,采用汇编语言编程有很多好处,它允许设计人员直接面向计算机资源,能够产生快速运行的高效代码。由于汇编语言直接与计算机结构相联系,采用汇编语言编程能帮助用户了解计算机的结构。汇编语言编程也有缺点,它相当慢、容易出错并且并不总是产生结构优化的程序。我们将在后续各章解决这个问题。现在,为了编写简单的程序并对微控制器有更多的了解,我们接下来将学习汇编语言。

4.1.2 采用汇编语言编写程序的流程

图 4-2 为采用汇编语言编写程序的实际流程。程序员用微处理器或微控制器汇编语言来编写程序,仅仅使用一个文本编辑器就可以编写。图 4-2 中的两行汇编程序来自 PIC 16 系列指令集,我们将很快认识它们。在程序员编写程序的计算机上运行交叉汇编器。从交叉汇编器的术语可以看出,计算机将代码汇编成另一种类型的代码,而不是计算机自身的代码。通常,交叉汇编器功能简化就成了汇编器。交叉汇编器汇编程序,也就是交叉汇编器将汇编助记符转换成为微控制器准备的机器码。在图 4-2 中可以看到,交叉汇编器将两行汇编代码转换成 PIC 16 系列的 14 位机器码字。大多数的微控制器都有专门的编程工具,能够从 PC 主机上下载机器码形式的程序,并把它写入微控制器的程序存储器中。



图 4-2 采用汇编语言编程

4.1.3 程序开发流程

采用汇编语言编写程序的流程需要放在更广的项目开发上下文环境中。图 4-3 显示的为一个简单嵌入式系统项目程序开发流程中的各个可能的步骤。程序员采用汇编语言编写程序，编写的程序称为*源代码*。然后运行在主机上的交叉汇编器来汇编该源代码。如果程序员可使用仿真器，那就可选择通过仿真器来测试程序。这样有可能发现程序错误，并根据错误对源代码进行修改，如此交替反复，直到程序无误。当程序无误时，程序开发人员通过和主机相连的独立"编程器"或者嵌入式系统自身的编程设备把调试好的程序下载到微控制器自身的程序存储器中。然后，程序员在实际的硬件上测试程序的运行情况。由于仿真器和硬件不同，程序在仿真器上正确不能保证程序在硬件上也一定正确，所以可能会出现错误，需要再次对源代码进行修改，如此交替反复。

68

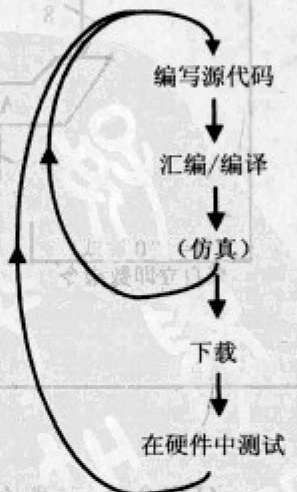


图 4-3 一个简单项目的开发流程

显然，即使是开发一个简单的项目，选择不同的软件工具也是有好处的。这些软件工具通常集成在一起，称为*集成开发环境*(Integrated Development Environment，IDE)。

4.2 PIC 16 系列指令集和 ALU

4.2.1 PIC 16 系列 ALU

在学习 16 系列指令集之前，有必要对 ALU 有更详细的了解（如图 4-4 所示）。明白 ALU 的工作原理能帮助我们理解指令集。从图中可以看到，ALU 对两个源数据进行运算。一个源数据来自 W 寄存器（也就是工作寄存器）。另一个源数据是立即数或来自数据存储器（Microchip 公司将数据存储器的存储单元称为“寄存器文件”）。立即数是与程序员写入程序的特殊指令相关的数据的一个字节。由此，我们就会想了解一些访问数据存储器的指令和一些在使用时需要指定立即数的指令。马上我们就会看到所有这些指令的例子。指令操作或使用的数据叫做操作数（operand）。操作数可以是数据，也可以是地址。我们会看到一些类型的指令总是需要指定它们的操作数，而另一些指令则不需要。

图 4-4 PIC 16 系列 ALU 框图

一旦指令执行，那么结果放在哪儿呢？Microchip 公司为多数指令提供了两种保存结果的方式，指令执行结果可保存在 W 寄存器中或者写回数据存储器。某些指令固定地选择一种保存方式，而另一些指令则根据指令中特殊 **d** 位的状态来确定采用哪一种保存方式。

4.2.2 PIC 16 系列指令集

我们现在开始学习 PIC 16 系列指令集，见附录 1。要想深入地理解 PIC 16 系列指令集，需花很长的时间去学习。附录 1 中，表格分为 6 栏，PIC 16 系列指令集含有 35 条指令，每条指令占据表格的一行。表格的第 1 栏为实际的指令助记符和指令使用的操作数的类型代码。共有以下 4 种操作数代码：

- **f** 是文件寄存器（也就是 RAM 中的存储单元）标识符，7 位；
- **b** 是位域标识符，选择所指定的寄存器文件中的位(bit)，3 位；
- **d** 是目标标识符，确定指令采用哪一种保存方式，1 位；
- **k** 表示立即数，如果为数据则为 8 位，如果为地址则为 11 位。

表格的第 2 栏概述了指令的功能。在某些情况下，这已给出了了解指令功能的足够多的信息。关于每条指令如何工作的更多更全面的描述见完整的微控制器数据手册[2.1]。表格的第 3 栏为指令执行需要的指令周期数。作为 RISC 处理器，我们希望每条指令都在一个节拍内完成。事实也是这样，从表中可以看到，除了那些造成程序分支的指令外，其他指令均在一个节拍内完成。我们在第 5 章讨论造成程序分支的指令的用法。表格的第 4 栏给出了每条指令实际的 14 位操作码。这是当交叉汇编器将用汇编语言编写的源程序转换为机器码时产生的代码。知道上面列出的操作数代码是怎样嵌入到操作码中是一件非常有趣的事。表格的第 5 栏是每条指令所影响的状态寄存器位（如图 2-3 所示）。

为了明白附录 1 中表格提供信息的含义，我们选择了 5 条指令作为例子来介绍。需要说明的是，汇编语言编程是不区分大小写的，本书中的所有例子均不区分大小写。所以，在不同的参考资料中指令助记符和操作数为大写和小写都是正确的。本书中为了格式上的统一，采用小写字母来书写汇编程序。现在请从附录 1 的指令集表格中找出下述指令。

- **clrw**——W 寄存器清零。不需指定操作数。表格第 5 栏指出状态寄存器的 **Z** 位受该指令的影响。由于该指令的结果恒为 0，所以状态寄存器的 **Z** 位总是设置为 1。状态寄存器的其他位不会受到影响。
- **clrf f**——标识符 **f** 指定的存储单元清零。清零哪个存储单元取决于程序员指定的 **f** 值。同样，由于指令结果为 0，状态寄存器的 **Z** 位受影响。
- **addwf f,d**——W 寄存器的值和标识符 **f** 指定的存储单元的值相加。选取哪个存储单元取决于程序员指定的 **f** 值。正如之前讨论的，指令执行的结果保存在什么位置可通过操作数 bit **d** 确定。由于指令执行的结果有不同的数值，3 个条件码位（也就是 **Z** 位、进位位 **C** 和数字进位位 **DC**）会受到该指令的影响。
- **bcf f,b**——清零存储单元的某一位。程序员需指定存储单元和要清零的那一位。位域标识符 **b** 值的范围为 0～7。没有任何状态寄存器标志受到影响，即使指令的结果是将存储单元置零。

70

□ **addlw k**——立即数值和 W 寄存器值相加。立即数值 **k** 须由程序员指定。结果保存在 W 寄存器中。和 **addwf** 一样，所有的条件码位都受该指令的影响。

4.3 汇编器和汇编格式

4.3.1 汇编器及 Microchip MPASM™汇编器简介

对于每个微处理器和微控制器，都有大量的汇编器或交叉汇编器可供使用。处理器厂商为了鼓励人们购买他们的产品，免费发布他们的汇编器。其他一些汇编器由软件专家工作室编写，通常很复杂，用于商业领域，需花钱购买。4.1.3 节中提到过，如今许多汇编器都集成到 IDE 中，成为 IDE 的一部分。本书使用 Microchip 公司提供的 MPASM 汇编器。MPASM 汇编器通常是作为 MPLAB IDE 的一部分来使用的。本章以及后续各章将会详细介绍 MPASM 和 MPLAB。

采用汇编语言编程，多数使用交叉汇编器，只有少数需要使用专用汇编器。

4.3.2 汇编格式

在学习了指令集之后，我们现在需要知道怎样将这些指令写入程序。还需要了解和掌握汇编程序的格式。汇编程序的格式很简单，如例程 4-1 所示。

71

例程 4-1 汇编格式

标号　助记符　操作数　注释

```
start bsf   status,5   ;select memory bank 1
      movlw B'00011000';config pattern for port A
      movwf trisa
      movlw 53
      ...
```

一行汇编代码中可能包含以下 4 个部分。

标号(Label)。每行的标号是可选的。如果给一行汇编程序指定标号，标号必须位于该行的最左边。汇编器将从标号的位置开始解释该行代码。一旦这样定义标号后，标号也可作为操作数来使用。标号必须以字母或下划线，而不能以数字开头。标号可以单独占一行，这种情况下，它用于标识其后面含有指令的汇编行。

指令助记符(Instruction mnemonic)。指令助记符来自指令集。除了最左边，指令助记符可以位于一行的任何位置。指令助记符和标号之间应至少有一个空格。

操作数(Operand)。操作数必须严格符合在指令集中指定的格式。为了更好地理解程序，操作数通常采用标号而非数字。如果有多个操作数，操作数之间应该用逗号隔开。

注释(Comment)。注释是可选的，用于给程序加入辅助信息，便于程序的理解，增

强程序的可读性。注释必须以分号开头，交叉汇编器会忽略每行代码中分号之后的部分。注释可位于每行指令的后面，也可单独占一整行。

汇编程序行可以包含一条按照上述格式定义的指令，也可以仅含有一条注释，或者完全空一行（这样有时可以方便代码的排列并提高程序的可读性）。

4.3.3 汇编伪指令

为目标微控制器编写的汇编程序首先必须经过汇编器的处理。为了使汇编器对汇编程序的处理更有效，灵活性更高，需要采取一种方法来传递信息和指令给汇编器，这些指令仅被汇编器识别，并引起汇编器做出相应的操作。这些指令称为汇编伪指令（Assembler directives），它们有各种不同的用途，例如定义目标处理器或指定程序在存储器中存放的位置。表 4-1 为一些 MPASM 例子。汇编伪指令用代码编写，看起来很像指令集中的助记符，但它们与助记符所起的作用却完全不同。

表 4-1　一些常见的 MPASM 汇编伪指令

汇编伪指令	作　用
list	通过参数项来设定汇编的一些信息*
#include	包含外部源文件
org	设定程序起始位置
equ	定义汇编常量；给标号赋值
end	程序块结束

* 参数项包含对数制和处理器类型的设定。

4.3.4 数的表示

72

在处理微控制器内部操作时，需要了解数是采取哪种进制来表示的。在汇编程序中，数有时用二进制来表示，有时用十进制来表示，有时用十六进制来表示，还有时候用八进制来表示。因此，对于汇编程序来说，能够识别和响应不同的数制是非常有益的。MPASM 汇编器首先把十六进制设置为默认数制。因此，如果程序员想只用（或主要用）十六进制来编写程序，那么汇编器会按照十六进制来解释所有的数。程序员想以另一种数制来表示数，就必须在数前面加上相应的前缀，如表 4-2 所示。在例程 4-1中，程序员采用默认的十六进制来编写程序。但是，在程序的第 2 行，程序员希望采用二进制来指定一个数，因此采用了表 4-2 中相应的二进制格式来表示这个数。在程序的第 4 行，他使用十六进制数 53 作为指令的操作数，由于十六进制是默认数制，因此不需要特别指明它的数制。

注意：十六进制数不能以字母开头，否则会被汇编器解释为标号。因此，以 a、b、c、d、e 或 f 开头的十六进制数的前面必须加 0。例如，数 $b2_H$ 必须写作 0b2。

表 4-2 MPASM 汇编器中数的表示

数 制	例 子
十进制	D255
十六进制	H8d 或 0x8d
八进制	O574
二进制	B01011100
ASCII	G 或 AG

4.4 编写简单的程序

一个简单的数据传送程序

下面我们来看一个简单的例程，该程序采用 MPASM 汇编器编写，使用 MPLAB IDE，如例程 4-2 所示。该程序是为电子乒乓球游戏硬件（附录 2）编写的，后面我们很快会用它来进行仿真。

73

例程 4-2 一个简单的数据传送程序

```
;*********************************************************************
;ELECTRONIC PING-PONG DATA MOVE
;This program moves push button switch values from Port A to the
;leds on Port B
;TJW 21.2.05                                       Tested 22.2.05
;*********************************************************************
;
;Configuration Word: WDT off, power-up timer on,
;                    code protect off, RC oscillator
;
     list p=16F84A
;
;specify SFRs
status   equ    03
porta    equ    05
trisa    equ    05
portb    equ    06
trisb    equ    06
;
      org    00
;Initialise
start  bsf    status,5        ;select memory bank 1
       movlw  B'00011000'
       movwf  trisa           ;port A according to above pattern
       movlw  00
       movwf  trisb           ;all port B bits output
       bcf    status,5        ;select bank 0
;
;The "main" program starts here
       clrf   porta           ;clear all bits in ports A
loop   movf   porta,0         ;move port A to W register
       movwf  portb           ;move W register to port B
       goto   loop
       end
```

程序以 5 行注释开始，每行都以分号开头。这些注释指明了程序的题目，简单介绍了程序的功能，给出了程序编写的时间和程序编写者。它们还给出了配置字的设置信息（如图 2-6 所示）和程序运行的基本条件，我们将学到将配置字的设置信息编程到微控制器中有多种方式。注释行之后是程序的第一个有效行，它使用 **list** 命令定义了要使用的微控制器。

接下来的一段程序使用 **equ** 伪指令定义了要使用的 SFR 存储单元。定义要使用的 SFR 存储单元在程序设计中是必需的，许多程序员对这一点感到吃惊。如果我们之前没有"告诉"汇编器处理器是什么，那么难道汇编器不应"知道"处理器是什么吗？答案是汇编器应当知道要面向的处理器什么，因此也要知道要使用的该处理器的存储单元是什么，所以我们必须提供这些信息。该程序仅使用了状态寄存器、端口 A 和 B、以及端口的控制寄存器 **TRISA** 和 **TRISB**。因而给它们指定了标号，标号的值来自图 2-5 中存储器映射的存储器地址。从 2.4.2 节中我们知道，存储区选择位在状态寄存器中。一旦 SFR 地址不包含此位（如图 2-5 所示），那么标号 **porta** 和 **trisa** 的值相同，标号 **portb** 和 **trisb** 的值也相同。在程序中，对每一对端口和控制寄存器（如 **porta** 和 **trisa**）仅使用一个标号，这对简化编程有一定的作用。在本例程中，我们不这样做，目的是为了在使用不同的存储单元时程序更加清晰。

在实际程序开始执行之前，有必要使用 **org** 伪指令指定程序的起始地址。对于起始地址，我们别无选择，它只能是复位向量地址，如图 2-4 所示。

后面的程序共使用了 7 条指令，除了一条分支指令外，其他 6 条指令都对数据的位和字节进行操作。这 6 条指令是：

- ☐ **clrf f**——将存储单元 **f** 清零；
- ☐ **movwf f**——将 W 寄存器的内容送至存储单元 **f**；
- ☐ **movf f,d**——如果 **d** 位设置为 0，将存储单元 **f** 的内容送至 W 寄存器；如果 **d** 位设置为 1，那么仅将存储单元 **f** 的内容送回存储单元 **f**；
- ☐ **movlw k**——将指令中 8 位的立即数 **k** 送至 W 寄存器；
- ☐ **bcf f,b**——将存储单元 **f** 的位 **b** 清零（也就是设置为逻辑 0）；
- ☐ **bsf f,b**——将存储单元 **f** 的位 **b** 设置为 1；
- ☐ **goto k**——程序跳转，从存储单元 **k** 中的指令开始执行。

org 伪指令之后的一段代码为初始化程序。这段程序配置了所使用的两个端口的每位的方向，需要访问端口控制寄存器 **TRISA** 和 **TRISB**。这两个寄存器位于 RAM 存储器的区 1 中，所以首先需要将状态寄存器的位 5 设置为 1（第 2 章已解释过）。在实际程序的第一行就执行这个操作，该行的标号为 **start**，使用 **bsf** 指令。**status** 标号在之前的程序中已经定义过，因此可以使用。如果之前没有定义，那么该行代码就必须这样写：

```
start      bsf     3,5      ;select memory bank 1
```

这种写法会使得程序不易理解。

从图 A2-1 中的电路图可以知道端口引脚的方向。从图中可以看到，两个按钮分

别连接到端口 A 的位 3 和位 4，相应地，这两位必须配置为输入。端口 A 的其他位均连接到 LED，因此这些位必须配置为输出。从 3.4 节可知，端口引脚为输出，那么相应的 **TRIS** 寄存器位必须为 0。端口引脚为输入，那么相应的 **TRIS** 寄存器位必须为 1。因此，我们必须将字 00011000 送至 **TRISA** 中。需要注意的是，TRISA 是 8 位，而端口 A 只有 5 位。所以需指定一个完整的 8 位的字送到 **TRISA** 中，即使其中有 3 位是不会用到的。由于没有一条指令能将一个字节的数据直接从程序中送至存储单元中，因此必须使用两行代码来完成这一传送过程。首先，使用 **movlw** 指令将需要的字 00011000 放在 W 寄存器中。在该过程中，使用二进制而不是默认的十六进制来表示字 00011000。然后，使用 **movwf** 指令将 W 寄存器的内容送至 **trisa** 中。采用相似的过程可配置端口 B。从电路图（图 A2-1）中可以看到，端口 B 的所有位均连接到 LED，所以端口 B 的所有位都必须配置为输出。因此，传给 **trisb** 的字为 00_H。初始化程序的最后一条指令是在状态寄存器中选择存储器的区 0，因为从这里开始程序将访问端口，而端口的地址在区 0 中。

最后的这段代码才是实现功能的有效程序。这段程序总共只有 5 行代码。程序持续地读入端口 A 的值并传送给端口 B。也就是说，如果按下两个按钮中的其中一个，那么与端口 B 的位 3 和位 4 连接的 LED 就会点亮。虽然端口 A 有 3 位配置为输出，但当读入端口 A 的值时，仍同时读取端口 A 的所有位。对于端口 A 的配置为输出的 3 位，读取的是内部数据锁存器的值（如图 3-11 所示）。因此，需使用 **clrf** 指令将端口 A 的所有位初始化为 0。

这段程序的实际数据传送部分为 **movf** 指令和 **movwf** 指令。**movf** 指令将端口 A 的值送至 W 寄存器中，然后 **movwf** 指令将 W 寄存器的值送至端口 B。**goto** 指令使程序跳转到前面定义的标号 **loop** 处，从而产生持续的循环。

75

值得注意的是，这里采用了 2 种不同的方法给标号赋值。**porta** 或 **portb** 这样的标号，由程序员通过 **equ** 伪指令给它们赋值。而 **loop** 这样嵌入到程序中的标号，由汇编器给它们分配数值。

整个程序以 **end** 伪指令结束。

4.5 使用开发环境编程

4.5.1 MPLAB 介绍

MPLAB 是一个集成开发环境（IDE），它可从 Microchip 公司的网站上免费下载[1,2]，本书的附属资源中也有一个 MPLAB 的副本。MPLAB 包含了采用汇编语言编写程序所有必需的软件工具，它可以将程序汇编并仿真，然后下载到编程器中。编程器可以自己制作也可购买现成的，或者将它设计在目标系统中。可以向 Microchip 公司或其他供应商购买更多更好的软件工具，然后集成到 MPLAB 中。这样，在使用 MPLAB 编程时，加上 MPLAB 自身已经提供的工具，就有更多的工具可供选择了。

MPLAB已经提供的工具有汇编器或仿真器，还有开发能力更强大的工具，如C编译器或仿真器驱动器。

MPLAB是一个可持续升级的软件包，有自己的用户手册[4.1,4.2]和在线帮助。因此，本书不是一本完整的MPLAB用户手册，而是对MPLAB用户手册的使用给出一个清晰的介绍，这样你就可以得心应手地使用它了。为了清晰地介绍MPLAB的用法，在本章和后续各章中采用了MPLAB 7.00版本和7.22版本的屏幕截图。

4.5.2 MPLAB的组成部分

MPLAB由许多不同的部分组成，这些部分协同工作构成了整个开发环境，它们包括以下几种。

- 文本编辑器(Text editor)。文本编辑器用于编写源代码。与记事本这样简单的文本编辑器相比，在某种程度上它提供更强大的文本编辑功能。它能识别正在使用的编程语言的主要部分。在汇编程序中，它用不同颜色对汇编程序的各部分作标记。用一种颜色标记指令，一种颜色标记标号，另一种颜色标记注释。这样，程序员就能一眼看出在汇编程序行中是否有错误。
- 项目管理器(Project manger)。在MPLAB中开发程序的最好方式是创建一个项目。MPLAB项目管理器将与一个项目相关的所有文件集合在一起，确保这些文件以适当的方式相互交流，并在需要时更新。
- 汇编器和连接器(Assembler and Linker)。汇编器的功能已经讨论过了。到目前为止，我们都假定只有一个源文件。然而，在高级项目中，程序代码由大量不同的源文件构成。连接器的作用就是把这些源文件合在一起，并给出每个源文件在存储器中的正确位置，以确保从一个源文件到另一个源文件的分支和调用能正确执行。
- 软件仿真器和调试器(Software simulator and debugger)。软件仿真器允许程序在主机的模拟CPU上运行，以此来测试已开发程序的功能。软件仿真器也可以模拟输入，观察输出值和存储器值。调试器允许对程序执行做全面的检测。例如，单步运行程序，或以较低速度执行程序，或在一个指定的位置停止程序。

76

表 4-3 一些 MPLAB IDE 中使用的文件扩展名

文件扩展名	功　能
.asm	汇编语言源文件
.err	出错文件
.hex	十六进制格式的机器码文件
.inc	汇编语言包含文件
.lib	库文件
.lst	绝对列表文件
.o	目标文件
.mcp	项目信息文件
.mcw	工作区信息文件

4.5.3 MPLAB 文件结构

即使创建一个简单的项目，在 MPLAB 中都会产生大量的文件。文件的类型由文件扩展名来指定，表 4-3 列出了一些 MPLAB 中使用的文件扩展名。只要创建一个项目，就会产生.**mcp** 文件和.**mcw** 文件。当使用汇编器时，编写的源代码文件的扩展名为.**asm**。源代码中可能含有.**inc** 文件，这将在第 5 章讲述。当源代码被成功汇编后，会产生输出文件：.**lst** 文件和.**hex** 文件。如果汇编出现错误，则会产生.**err** 文件。

4.6 MPLAB 指南

通过学习本节，你将会熟悉创建一个项目的所有步骤，包括编写源代码和将源代码汇编产生输出文件。如果你工作或学习的地方没有 MPLAB，为了学习本节内容，请下载并安装最新版的 MPLAB。

打开 MPLAB IDE，会看到如图 4-5 所示的程序界面。如果同时也打开了一个空白的输出(Output)窗口，将它关闭。除了屏幕左上角的工作区(Workspace)窗口外(工作区窗口不能关闭)，主屏幕的其他地方都是空白的。

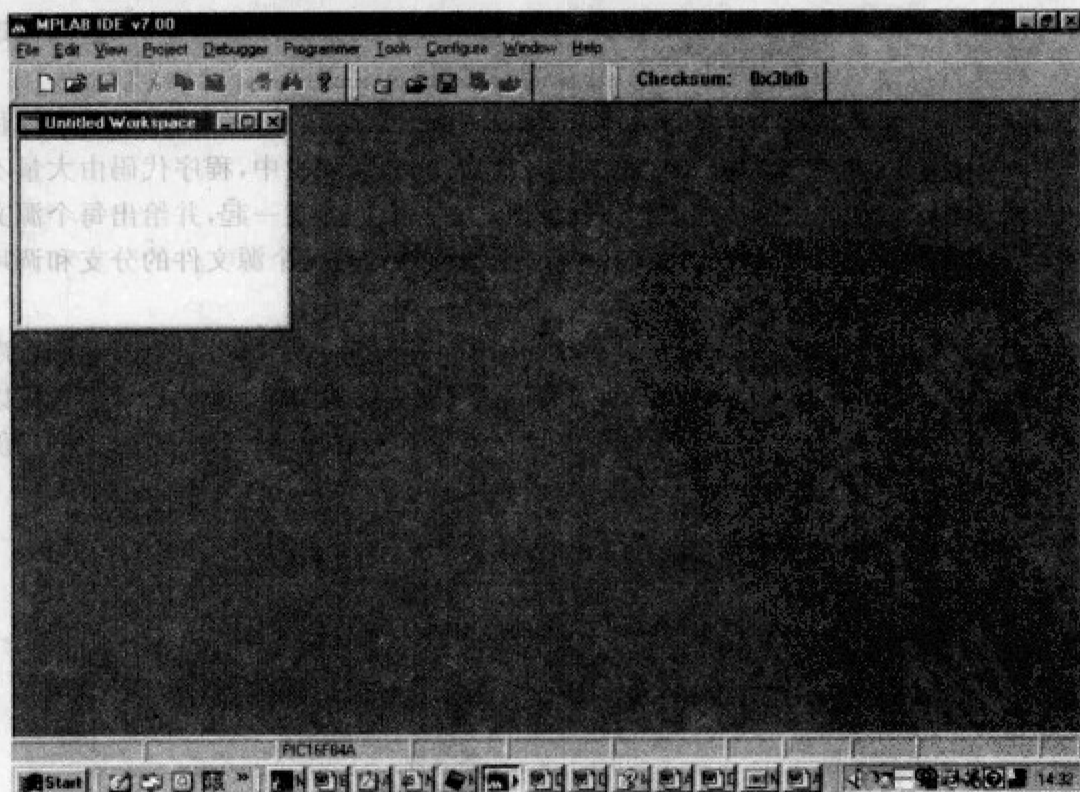


图 4-5 MPLAB IDE 屏幕

4.6.1 创建项目

单击菜单栏上的 Project 按钮，弹出下拉菜单，如图 4-6 所示。

创建一个项目有两种方式，都可通过此下拉菜单实现。一种方式是使用 **Project Wizard**，另一种方式是选择 **New**…。使用 **Project Wizard** 创建项目时，会有一系列的对话框，在对话框中按照下述要求进行选择。

77

Device(器件)：**PIC16F84A**

Active Toolsuite(可用的工具集)：**Microchip MPASM Toolsuite**

显示的 Toolsuite contents(工具集内容)有：

MPASM Assembler

MPLINK Object Linker

MPLIB Librarian

Project Name(项目名称) <自己指定>

Project Directory(项目目录) <自己指定>

Add existing files(添加存在的文件)…不添加

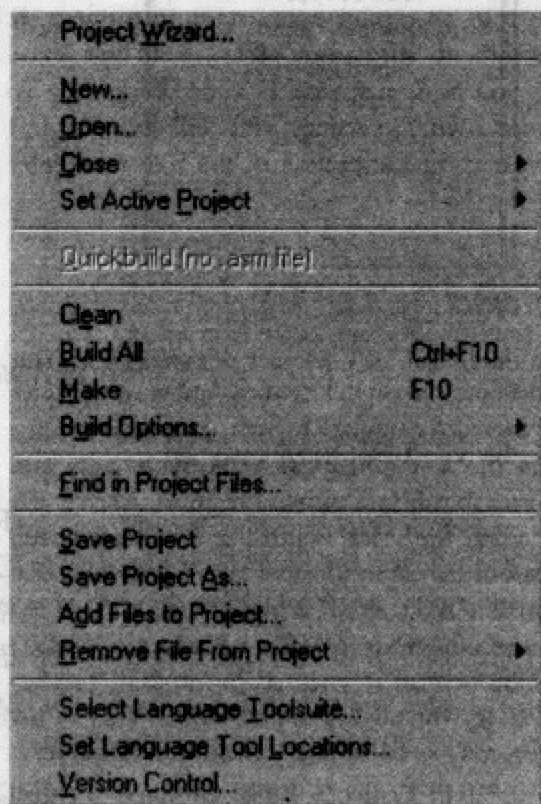


图 4-6 Project 下拉菜单

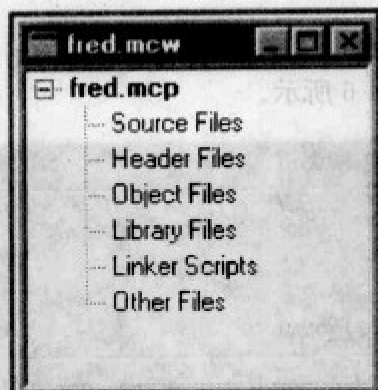
当单击 **Finish** 时，Workspace 窗口会刷新，并显示你所选择的文件名，如图 4-7a 所示，图中的项目叫做 **fred**。

78

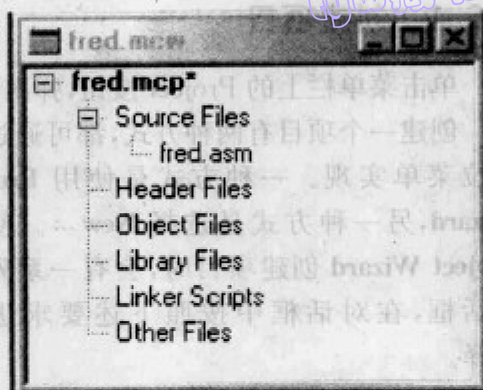
4.6.2 编写源代码

现在单击 **File**>**New** 打开一个新文件，将例程 4-2 的程序输入进去。输入几行程序后，使用 **File**>**Save As**…保存文件。选择文件类型为 **Assembly Source File**，并另存为<你的项目名称>**.asm**。继续输入代码，你会发现 MPLAB 已经认为该文件是一个汇编源文件(Assembly Source File)了。MPLAB 使用不同的颜色标记标号、指令助记符、数值数据、汇编伪指令和注释。程序输完后，再打开 Project 菜单，单击 **Add Files to Project**…，然后选择刚才你保存的文件。你的工作区现在就会变成图 4-7b 所示的样子，当然显示的文件名是你自己设定的文件名了。现在你就会发现 Workspace 窗口变得丰富了，这对项目管理非常有帮助，因为窗口中完整地列出了与项目相关的所有文件。

79



(a) 新创建的项目



(b) 加入源代码的项目

图 4-7 Workspace 窗口

4.6.3 对项目进行汇编

现在要做的是项目开发过程中测试环节的一步。你已经编写了新的源代码，现在需要知道编写的源代码是否能被正确地汇编。汇编器会对你的代码做一系列的检查，如果汇编格式、指令助记符、标号或其他格式使用有误，汇编器会返回错误。但是，须注意的是汇编器只能有效地检查程序在语法上是否正确，但不能确保程序的可行性。除了上面提到的，汇编器也不知道目标硬件，不知道在汇编程序中是否指定了微控制器。所以，程序汇编正确并不能保证程序一定能正常运行。

单击 **Project>Build Options>Project>MPASM Assembler** 检查是否正确设置了默认数制，确保在对话框中选择 **Hexadecimal**。在同一个对话框中，你还可以选择是否区分源代码的大小写。如果你已直接将例程 4-2 的代码拷贝到项目中，不必考虑这一点，因为例程 4-2 的代码全是小写的。但在以后的编程中，你会用到这一点[①]。

单击 **Project>Build All** 调用 MPASM 汇编器。在需要更新文件时，单击 Build All 也能更新所有的文件。调用 MPASM 汇编器会打开 **Output** 窗口，Output 窗口报告构建(build)程序的进程。在 Output 窗口中，你会得到"Build succeeded"或"Build failed"信息。构建成功时，会闪现一个有绿色横条(表示成功)的窗口；构建失败时，会闪现一个有红色横条(表示出错)的窗口。

无论程序的构建是成功还是失败，都请打开文件<你的项目名称>. **lst**。该文件在你为项目指定的目录下。单击 **File>Open**，在对话框的 **Files of Type** 中选择 **All Files**。. **lst** 文件包含了非常详细的信息。. **lst** 文件含有源代码，每条源代码旁边都有该条代码汇编后的机器码，同时还给出了汇编器生成的错误和警告信息。例程 4-3 是例程

① 虽然汇编语言编程不区分大小写，但是在对话框中选择区分大小写，能使你有好的程序风格，程序可读性强。——译者注

4-2 的列表(list)文件的部分内容。注意机器码是怎样表示的,还应该注意在主程序之后还有一些必要的基本程序信息。

80

例程 4-3 Data_Move 列表文件的部分内容

```
                    00029        ;The "main" program starts here
0006    3000        00030          movlw 00    ;clear all bits in ports A and B
0007    0085        00031          movwf porta
0008    0086        00032          movwf portb
0009    0805        00033   loop   movf     porta,0 ;move port A to W register
000A    0086        00034          movwf    portb   ;move W register to port B
000B    2809        00035          goto     loop
                    00036          end

MPASM 03.90 Released                          DATA MOVE.ASM   3-10-2005
15:55:03          PAGE 2

SYMBOL TABLE
  LABEL                          VALUE

__16F84A                        00000001
loop                            00000009
porta                           00000005
portb                           00000006
start                           00000000
status                          00000003
trisa                           00000005
trisb                           00000006
.....etc
```

一旦出错,那么在列表文件中就会有一个错误编号和一条错误信息。大多数情况下,简单的字母或单词输入错误是很容易找到的,并且只需改正源代码再构建程序就可以将它修复。但是如果错误很难理解,就需要在 Help 菜单或参考文献 4.1 中根据错误编号查找该条错误的详细信息。最后,单击 **Project>Close** 关闭当前项目。

一旦正确的源代码已经构建,接下来需要做的就是将程序下载到微控制器的存储器中或者对程序进行仿真。接下来我们将学习如何对程序进行仿真。

4.7 程序仿真简介

本节介绍了 MPLAB 仿真器——MPSIM™,并对我们刚刚汇编的程序进行了仿真。

4.7.1 开始仿真

在 MPLAB 中,单击 **Debugger>Select Tool>MPLAB SIM** 调用仿真器。单击菜单栏上的 **Debugger** 按钮,弹出仿真器下拉菜单,如图 4-8 所示。

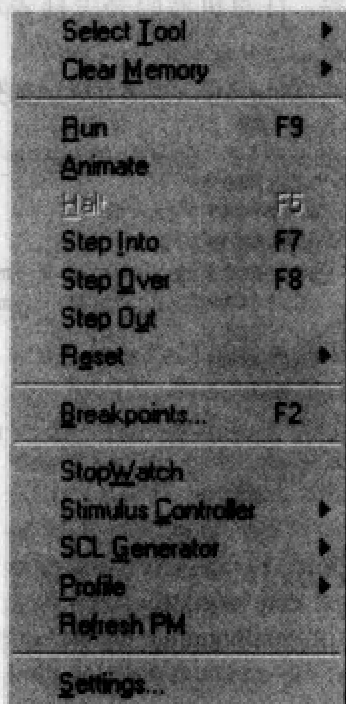


图 4-8 仿真器菜单

4.7.2 产生端口输入

该程序用于乒乓球游戏硬件，因此我们需要在端口 A 的引脚 3 和引脚 4 上创建游戏中两个乒乓球拍的仿真输入。有两种方式来产生仿真输入，采用哪一种方式取决于输入是否与指令执行同步。我们采用较简单的一种方式——用户控制下的异步输入。

81

单击 **Debugger>Stimulus Controller>New Scenario**，会弹出一个对话框。在对话框中你可以给端口引脚配置不同的输入类型，配置好后，单击 **Fire** 按钮让仿真器接受你的配置信息。在 **Pin** 下选择 **RA3**，在 **Action** 下选择 **Toggle**；然后选择 **Pin** 下的 **RA4**，同样再在 **Action** 下选择 **Toggle**。关闭项目时，配置信息会另存为. **stc** 文件。

4.7.3 观察微控制器各部分状态

在程序仿真过程中，可查看微控制器各部分的状态，包括程序存储器、SFR、数据存储器等等。打开 **View** 菜单，从中可为微控制器的各部分打开一个观察窗口。但是如果这样做，整个屏幕就会堆满了各个观察窗口，显得比较混乱。在 **Watch** 窗口中，你可以选择只观察你想观察的变量，对其他变量不予考虑。单击 **View** 下拉菜单中的 **Watch** 选项，就会打开一个 **Watch** 窗口，然后选择 **PCL**、**TRISA**、**PORTA**、**TRISB** 和 **PORTB**。图 4-10 中左上角的窗口就是 **Watch** 窗口。

4.7.4 程序复位和运行

单击 **Debugger>Reset** 或按 F6 键可以复位仿真 CPU。选择前一种方法，可以有 4 种 CPU 复位方式，它们分别与 2.8 节中所提到的 PIC 微控制器的复位过程对应。如果想简单地复位仿真 CPU，不去考虑具体复位方式的各种情况，可单击调试工具栏上的 **Reset** 按钮(图 4-9)。如果 MPLAB IDE 中没有调试工具栏，可选择 **View>Toolbars>Debug**。

82

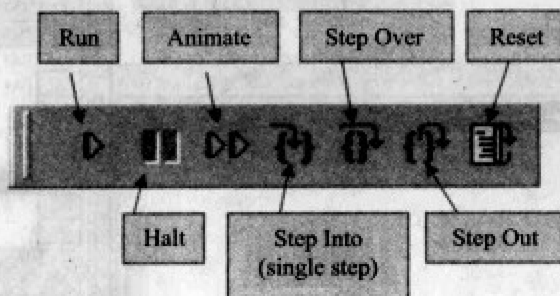


图 4-9 MPSIM 软件仿真器的调试工具栏

运行程序有 3 种不同的方式。每种方式都可以在 Debugger 下拉菜单下选择，或者直接单击调试工具栏上的相关按钮。这 3 种方式如下所示。

- ❑单步运行(Single Step)。你可一步一步地运行程序，每步只执行一条指令。此版本的 MPLAB 将这种模式称为 **Step Info**。
- ❑连续单步运行(Animate)。相当于自动执行的单步操作。程序连续不断地运行，运行较慢。每执行一条指令后，屏幕就会刷新一次。程序运行的速度可通过调用 **Debugger>Settings>Debugger Animation** 来设定。
- ❑连续运行(Run)。运行程序，但在程序运行时不会更新屏幕上显示的窗口。它可以接受输入激励。

在调试程序时，可能需进入或退出一个子例程，分别对应调试工具栏上的按钮 **Step Over** 和 **Step Out**。这 2 个功能对延时程序的调试很有用。延时程序的作用是使程序延时一段较长的时间后再执行，在这段时间内程序不做任何事情，因此在仿真器上调试时不需要关心这段仿真过程，选择 **Step Out** 退出延时程序，继续执行后面的程序。

如果你已经完成上述所有步骤，你的计算机屏幕会类似于图 4-10，当然你的 MPLAB 中各个窗口的排列有可能和图 4-10 不同。Wacth 窗口位于屏幕的左上角，激励控制器位于屏幕右上角，而源文件位于屏幕的左下角。此时仿真 CPU 处于复位状态，源文件中的箭头表示程序计数器，此时正指向程序的第一条指令。查看 Watch 窗口中 **PCL** 的值也可以看出程序计数器指向第一条指令[①]。

① 因为程序的第一条指令的地址始终是 0，此时 PCL 的值为 0x00，说明 PC 指向的是第一条指令。——译者注

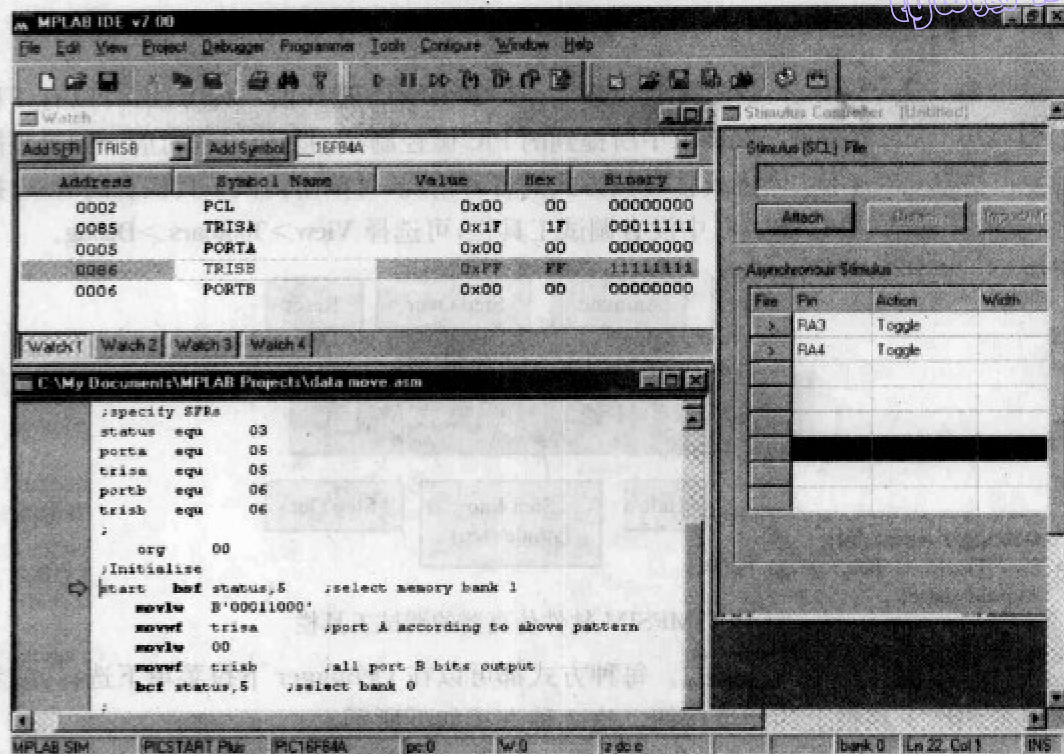


图 4-10 一个用于简单仿真的 MPSIM 组成

重复单击 **Step Info** 按钮，单步运行程序。程序首先会一步一步地执行初始化程序，你会发现 Watch 窗口中的 SFR 值发生了变化，并且 **PCL** 值也在递增。源程序的最后 3 条指令构成一个循环，程序在此处将反复执行。现在给 RA3 和 RA4 赋值。此时显示窗口中 PORTA 的值并未更新为你设置的值，在下一条指令执行时，才会更新。继续单步运行程序，端口 A 和端口 B 的值（PORTA 和 PORTB 变量）更新为你设置的值。现在试着单击 **Animate** 按钮程序。此时，程序会连续不断地运行，但在程序运行时，仍可以响应输入激励[①]。

4.8 下载程序到微控制器[②]

83

大多数现代微处理器都有采用 Flash 技术制作的片上程序存储器。对微控制器编程的过程需要在精确时序下把数据传送到芯片内部，还需要一定的编程电压，该电压通常比标准的电源电压要高。因此，微控制器的某些引脚还需具备一个功能，即在编

---

① 即你可以在程序运行过程中给 RA3 和 RA4 赋值。——译者注

② 下载程序到微控制器也称为对微控制器编程。不光是针对微控制器，对于所有带有存储器的硬件电路来说，凡是将程序写入存储器的过程，都称为下载程序到该硬件电路，也称为对该硬件电路编程。——译者注

程模式下将程序数据传送到芯片上和传输编程电压。

过去,对带有存储器的IC(或一个独立设备,或一个有存储器的微控制器)编程需要将它放在编程器中。将编程器连接到桌面计算机上,才能开始编程。然而,随着存储器技术的提高,对微控制器编程变得简单,将必须的编程电路设计到目标系统中也非常容易。这就意味着,现在多数微控制器可以不需要编程器,直接在目标系统上编程了。在后续各章将会学到这种技术,本章中我们仍采用传统的方法(即使用编程器)来对微控制器进行编程,这就需要将微控制器从目标电路中取下来并放在编程器中。

84

Microchip公司提供了一个很受欢迎并且价格低廉的编程器——PICSTART® Plus,如图4-11所示。许多硬件编程设计(主要指设计源代码)都可以通过该编程器下载到微控制器中,包括网络上共享的许多个人设计程序。PICSTART编程器通过一条串行电缆和主机相连,MPLAB集成开发环境中有软件负责与PICSTART编程器通信。PICSTART编程能够对引脚为18到40的众多双列直插式封装的微控制器编程。加上适配器后,就能对其他封装形式的微控制器进行编程。

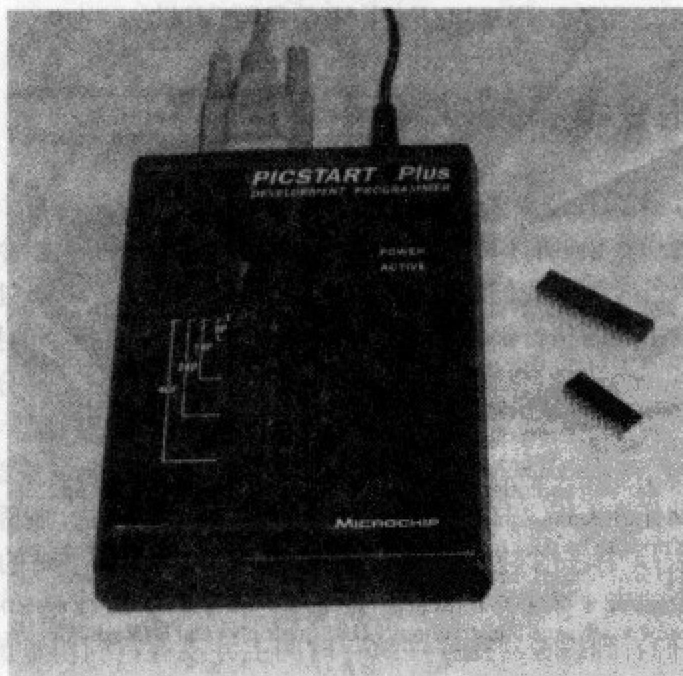


图4-11 PICSTART Plus编程器

下面的内容教你怎样通过PICSTART Plus编程器下载代码到微控制器。如果你有一个PICSTART Plus编程器和乒乓球游戏硬件,就可以立即将之前已经创建好的程序下载到乒乓球游戏硬件的微控制器中。

首先需要给PICSTART编程器接上电源,并将它连接到你电脑的串行端口上。在MPLAB IDE中,选择**Programmer>Select Programmer>PICSTART Plus**。然后选择**Programmer>Enable Programmer**来启动编程器。可以从Output窗口中看到编程器是否启

动。如果有问题，需要检查 **Programmer>Settings>Communications** 的设置是否正确。如果你的 MPLAB IDE 中没有编程器工具栏，选择 **View>Toolbars>Picstart**。

85

打开 PICSTART 编程器上的零插拔力（Zero Insertion Force，ZIF）插槽。将 16F84A 放在插槽中，按照编程器上的图例，确保芯片放在正确的位置。然后关上 ZIF。在 MPLAB 中打开你的项目，现在你就可以使用图 4-12 中所归纳的功能对 16F84A 芯片进行编程了。

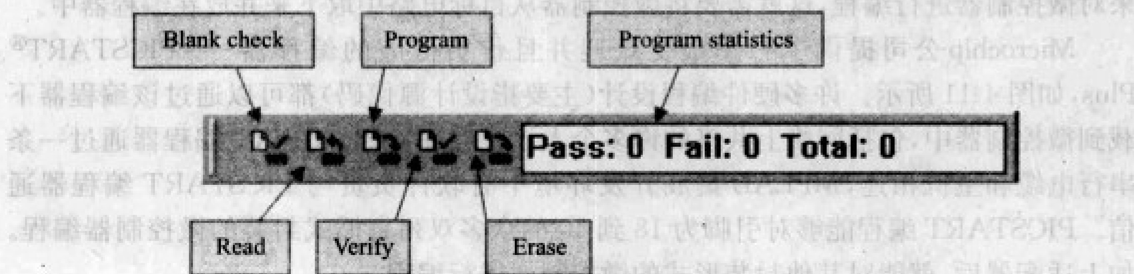


图 4-12　MPLAB 编程器工具栏

4.9　CISC 指令集和 RISC 指令集比较

由于 Atmel 8051 微控制器是 CISC CPU，我们希望该微控制器与 16 系列 RISC 微控制器相比有更大的指令集，而且其中一些指令的功能更强大。扩大指令集和增强指令功能会增加指令的执行时间。我们的这种想法被证实是正确的。8051 处理器核有 111 条指令。其中大多数指令的执行都在一个指令周期内完成，每条指令的执行都需要 12 个振荡器周期（与之相比，一个 PIC 指令周期只需 4 个振荡器周期）。部分指令需要 2 个指令周期，2 条"高级"指令 **MUL**（乘法指令）和 **DIV**（除法指令），每条则需 4 个指令周期。

从 RISC 的角度来看，CISC 指令集提供了多种强大的功能。需要数条 RISC 指令才能完成的简单动作在 CISC 中缩减到只需 1 条指令就能完成。例程 4-4 为 2 个简单的例子。在例程 4-4a 中，一个字节的常量数据[①] 送至叫做 **mem_loc** 的存储单元中。这需要 2 条 PIC 指令或者 1 条 8051 指令。编程器得到的好处仅仅是 8051 程序代码行要少些，但事实上我们看到编程器在程序执行时间上并没有得到明显的好处，因为 2 条 PIC 指令需要 2 个指令周期，而一条 8051 指令也需要 2 个指令周期。确实，由于 8051 指令的指令周期更长些，因此最终 PIC 微控制器指令在时间上更有优势。

有趣的是，在例程 4-4b 中，PIC 微控制器指令就没有明显的时间优势了。PIC16 系列只有条件跳转指令而没有分支指令。因此，要实现条件分支，须在跳转指令之后加上一条 **goto** 指令。对于 PIC 微控制器，如果要执行条件分支则需要 3 个指令周期，而对于 8051 来说，只需要 2 个指令周期。8051 只需要一行代码就可以实现条件分支。

① 在 PIC 中叫做 *literal data*，在 8051 中叫做 *immediate data*，而在中文中都翻译为立即数。——译者注

现在，只要两者的指令周期相同，那么 8051 CPU 就更有时间优势。

86

例程 4-4 RISC 和 CISC 比较

```
movlw   22          ;1 cycle              mov mem_loc,#22     ;2 cycles
movwf mem_loc       ;1 cycle
```

(a) 将立即数送至存储单元

```
btfsc status,0      ;1 cycle (no skip)    jc new_place        ;2 cycles
goto new_place      ;2 cycles
```

(b) 如果进位位置 1，分支

RISC 处理器比 CISC 需要更多的代码行，这并不是一个大的缺点。在后续两章，我们将知道这个缺点甚至是可以改善的。在汇编程序中使用宏，使用 C 语言这样的高级语言编程，程序员就不再直接关心汇编代码行数了。

4.10 更多的了解——16 系列指令集格式

在这里我们花点时间来进一步学习指令码的组成部分，在第 1 章介绍 12F508/9 时已经讨论过这些内容。PIC 16 系列的指令有 4 种可能的指令字格式，如图 4-13 所示。指令字为 14 位，从程序存储器中得到，沿程序总线送至各个需使用指令字的模块（如图 2-2 所示）。14 位指令字的形式如图 4-13 所示，从位 0 到位 13。操作码是指令字中实际的指令部分，它总是位于指令字的高位。操作码送入“指令译码和控制”单元（如图 2-2 所示），但是所有操作码并不总是有相同的长度。

87

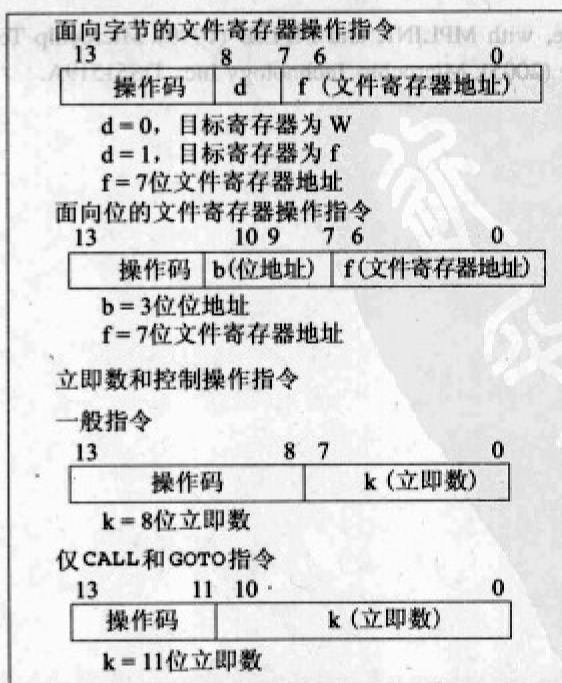


图 4-13 PIC“中端”微控制器的指令格式

如果指令包含一个寄存器文件地址，那么它就是图 4-13 中的第一个格式。在该格式中，最高 6 位的为操作码，而最低的 7 位用于保存寄存器文件地址。这 7 位送至“直接地址”总线上（如图 2-2 所示）。事实上，由于 F84A 只有小容量的存储器，因此只用到指令字中最低的 5 位，从图 2-2 中的“直接地址”总线的大小就可看出。位 7 保存 d 位。不同类型的指令使用不同的指令字格式，从图 4-13 中可以知道哪些指令采用哪种格式。

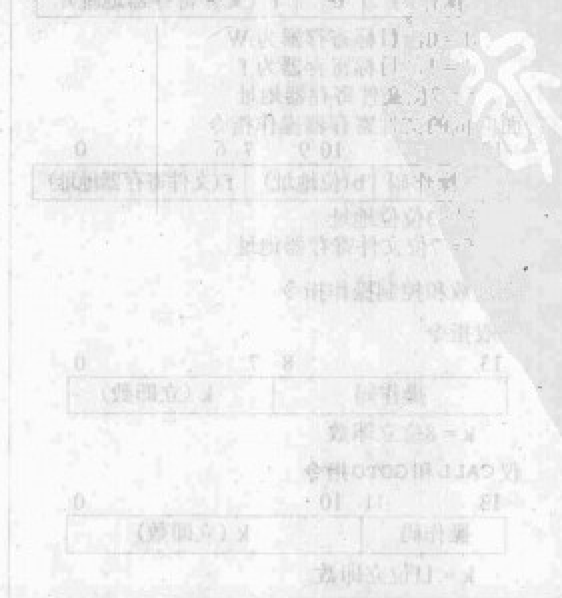
小结

如果你理解了本章的内容，那么现在你已经在嵌入式系统方面取得了很大进步——开始成为一名嵌入式系统程序员了。本章的重点如下所述。

- ❑ 汇编语言作为一种编程语言，是嵌入编程中所使用的编程工具的一种。它有自己独特的代码规则和程序技术。
- ❑ 在开发程序时，有必要学习和采用 IDE。MPLAB IDE 是针对 PIC 微控制器的一款优秀的开发工具，适用于学习者，也适用于专业人员，并且它是免费的。
- ❑ 如果有人迫切地想将程序放进微控制器中运行，那么学习仿真器的功能是非常有用的。MPLAB 中的仿真器能让用户快速地测试程序行为，该仿真器是一款非常优秀的学习工具。

参考文献

4.1. MPASM User's Guide, with MPLINK and MPLIB (1999). Microchip Technology Inc., DS33014G.
4.2. MPLAB User's Guide (2005). Microchip Technology Inc., DS51519A.



第5章 创建汇编程序

第4章介绍了汇编语言编程的基本规则和PIC® 16系列指令集的一些指令。通过对第4章中简单汇编程序的学习，我们已经具备了创建汇编程序的基本能力。现在需要进一步提高编写汇编程序的能力，那么我们首先需要考虑的是由程序块组成的汇编程序的结构是怎样的。因此，现在需要学习有关汇编程序结构的基本知识，只有这样才能在实际中编写出功能正确可靠的结构化程序。

本章中你将学到：

- ❑ 如何形象化一个程序并通过程序框图来表示它；
- ❑ 如何使用子例程；
- ❑ 如何实现延时；
- ❑ 如何使用查找表；
- ❑ 逻辑和算术指令；
- ❑ 如何简化和优化汇编语言编程；
- ❑ 软件仿真器的更多高级功能。

5.1 创建结构化程序概述

设计程序并不是一个简单的工作，需要在开始编写代码前就考虑和规划程序结构，这一点很重要。在采用汇编语言编写程序时这点尤其重要——第4章中曾提醒过大家，汇编语言编程的一个问题是容易导致非结构化的、"意大利面条"式的程序。因此，我们必须学会通过程序框图来表示程序，下面是2个平常家用产品的例子。

5.1.1 流程图

流程图是一种非常好的程序框图。流程图中有许多种符号，用于表示程序的各个部分。通常我们只需使用其中的两种符号就可以完成一个好的流程图了，它们是表示过程或动作的矩形框和表示判断的菱形框。

图5-1为一个简单的电冰箱控制器流程图例子。用户通过单一的控制(一个可调的电位计)来设置想要的温度。在电冰箱内，有一个温度传感器。温度通过启动或关闭压缩机来控制。当压缩机工作时，冰箱内温度降低，程序同时读取实际温度和需要

的温度，并判断哪个温度更高。如果实际温度更高，就会启动压缩机。如果两者的温度相差很大，就会发出警告。图 5-1 中的流程图仅采用矩形框和菱形框，表示了这一过程。从图中可看到，每个表示判断的菱形框中都含有一个问题，每个问题都有“是”和“否”两种回答。对于这两种回答，菱形框有两个程序出口。从图中还可看到，整个程序无限循环。这是一个常见的嵌入式系统程序结构，有时也称为超循环(super loop)。

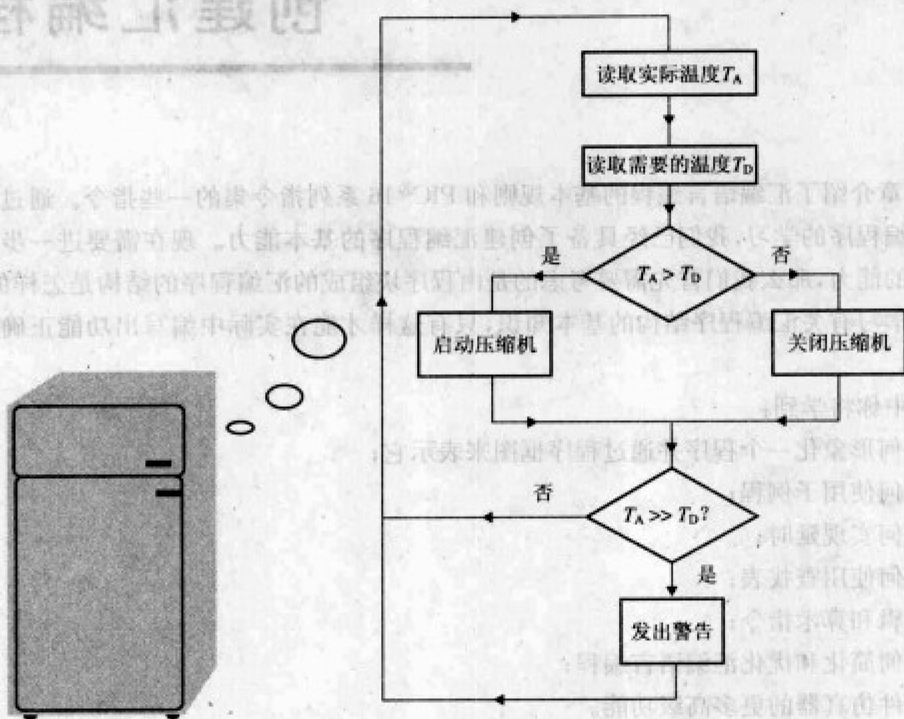


图 5-1 简单的电冰箱控制器流程图

流程图可以画得非常详细也可以画得很简单，通常流程图只要画到不需太难就能转换为汇编程序就足够了。在某些情况下，可以只画出流程图的整个框架，而用子流程图来分别表示内部的各个部分。

有些人认为流程图是一种过时的技术，就像汇编程序本身也是一种过时的编程语言一样，它不能产生结构化的程序。然而，流程图易于学习和使用，并且对于一些简单的程序思想，它能够清晰地表示出来。因此，基于这个原因，我们在后面还是使用它来设计程序。

5.1.2 状态图

在流程图中，程序通过一系列动作或事件来表示，这一系列动作或事件的发生就代表了程序的执行过程。然而，许多产品有另外一种不同的程序行为。程序是从一个状态转移到另一个状态，可能会在一个状态上停留很长时间，只有当这段时间完成后或者一个特定事件发生时才离开这个状态。对于这些产品，它们的程序最好通过状态

图(state diagram)来表示。状态图是不同于流程图的另一种程序框图。同流程图一样,状态图有许多符号和各种形式,在使用时很复杂。但是,我们的目的是学会用状态图来表示程序,所以不必使用状态图的所有复杂的符号,而只是用有标记的圆圈表示一个状态并用箭头表示状态间的连接就可以了。状态图显示了在什么条件下会发生状态的迁移。每个箭头都标有造成状态变化的条件。

图 5-2 是一个简单的状态图,表示的是一台家用洗衣机的功能。在启动时,洗衣机首先进入"就绪"状态。如果洗衣机门关上且用户开始洗衣,那么洗衣机开始加水,有一个水位传感器用于检测加水是否完成。同时,洗衣机也会计算加水的时间,如果没有在规定的时间内完成加水,会进入"出错"状态,报告错误。造成这种情况可能是由于水压不够或者阀门失效。采用超时来判断是否在规定的时间内完成加水比用传感器来检测水流或水压更方便,代价更低。在加水之后进入"加热"状态。同样,如果水没有在规定的时间内加热,将发生超时,进入"出错"状态。洗衣过程就如图 5-2 中所示依次进行。每个状态都有两个退出条件,"成功"退出条件表示该状态功能完成,进入下一状态;而另一个退出条件表示该状态功能未完成,进入"出错"状态。

91

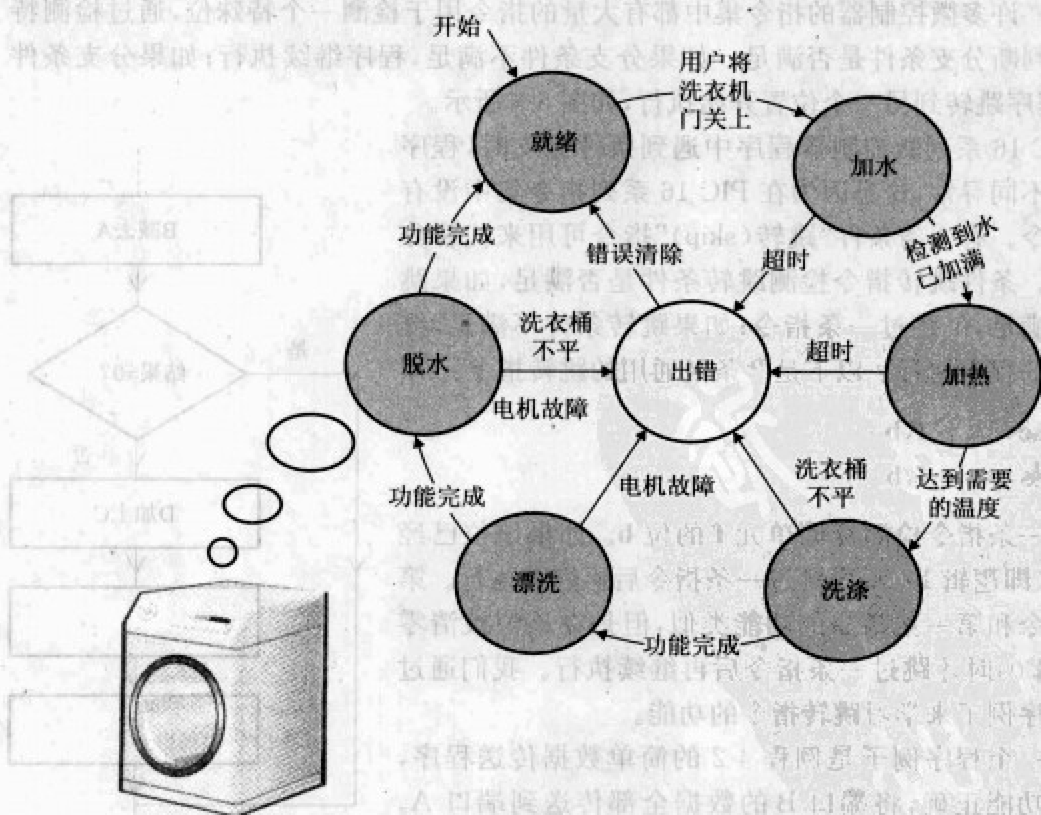


图 5-2 洗衣机控制程序的状态图

从编程的角度看,状态图比流程图更抽象,不易直接转换为汇编代码。事实上,将每个状态转换为状态自身的流程图通常是很有用的。为了保持清晰的结构和确保好

的编程实现，应当清晰地定义状态在何时进入和在何时退出。在本章后面的电子乒乓球游戏程序中，将同时使用流程图和状态图来表示程序的结构。

5.2 流程控制——分支和子例程

如图 5-1 和图 5-2 所示，很少有程序能以一条连续完整的指令序列来执行。将程序执行从一个程序段移动到另一个程序段的技术统称为*流程控制*(flow control)。流程控制是本节讨论的内容。我们已经学习了 PIC 的 **goto** 指令，**goto** 指令无条件地将程序执行从程序的一个位置移到另一个位置。现在，我们来学习如何使用条件分支和子例程。

5.2.1 条件分支和位操作

微处理器或微控制器程序的最重要的特征之一是能够做出"判断"，也就是说能够根据逻辑变量的状态执行不同的功能。这些逻辑变量通常是条件码或状态寄存器中位的值。许多微控制器的指令集中都有大量的指令用于检测一个特殊位，通过检测特殊位来判断分支条件是否满足。如果分支条件不满足，程序继续执行；如果分支条件满足，程序跳转到另一个位置开始执行，如图 5-3 所示。

PIC 16 系列微控制器程序中遇到条件分支时，程序的执行不同寻常，这是因为在 PIC 16 系列指令集中没有分支指令。有 4 条条件"跳转(skip)"指令可用来代替分支指令。条件跳转指令检测跳转条件是否满足，如果跳转条件满足，仅跳过一条指令；如果跳转条件不满足，继续正常的程序执行。以下是 2 条最通用的跳转指令：

```
btfsc    f,b
btfss    f,b
```

第一条指令检测存储单元 **f** 的位 **b**。如果该位已经被置位(即逻辑 1)，程序跳过一条指令后再继续执行。第二条指令和第一条指令的功能类似，但是在该位被清零(即逻辑 0)时才跳过一条指令后再继续执行。我们通过一个程序例子来学习跳转指令的功能。

第一个程序例子是例程 4-2 的简单数据传送程序，该程序功能正确，将端口 B 的数据全部传送到端口 A。但是在这里，假设不想将端口 B 的所有数据都传送到端口 A，而只想传送其中的一位，或者将端口 B 的一位传送给端口 A 的另一位，那么可以使用 16 系列指令集中的"面向位"的文件寄存器操作指令，总共有 4 条：**btfsc**、

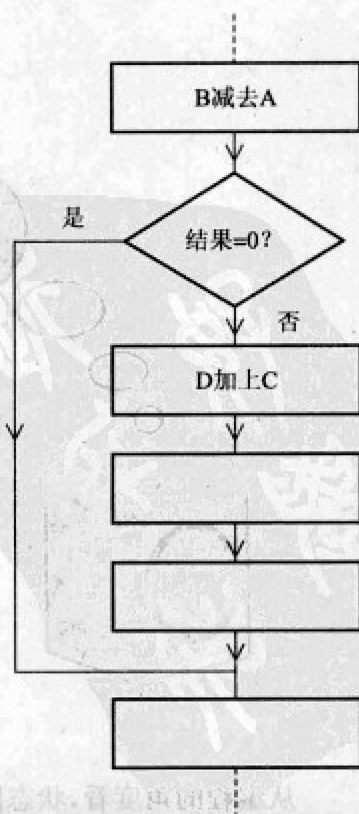


图 5-3 条件分支

btfss(这两条刚刚学过)、**bsf** 和 **bcf**。

例程 5-1 的程序与例程 4-2 的程序有几乎相同的功能,但是例程 5-1 中使用的是位操作指令。由于仅对单一的位进行操作,因而对端口写时不会影响到端口的其他位。如果与该位相关的微开关闭合,LED 点亮。但是,即使是这种简单的任务也要仔细地思考。当按钮按下时端口输入变为低电平,此时程序需要设置输出位(去点亮 LED);如果按钮未按下,端口输入为高电平,此时需将输出位清零。在这里就含有一个程序的选择过程——在高级语言中我们将此称为 *if…else* 结构。跳转指令功能简单,因而单靠跳转指令还不能达到"选择"这一功能。一个变通的方式是先给输出位预置一个值,然后如果在程序中发现这个值的设置有误再更改它。

93

例程 5-1　对单一位的检测和操作

```
;The "main" program starts here
        movlw 00            ;clear all bits in port A and B
        movwf porta
        movwf portb
loop    bcf   portb, 3      ;preclear port B, bit 3
        btfss porta, 3
        bsf   portb, 3      ;but set it if button pressed
;
bcf   portb, 4     ;preclear port B, bit 4
btfss porta, 4
bsf   portb, 4     ;but set it if button pressed
goto  loop
end
```

编程练习 5-1

新建一个项目,取名为 Bit_Set,或者自己取一个名称。将例程 4-2 作为源文件复制到项目中,用例程 5-1 替换程序中的主要代码段。构建项目并仿真。使用 Stimulus Controller 为端口 A 的引脚 3 和引脚 4 产生输入信号,在 Action 中选择 Toggle。打开 Watch 窗口,选择 **PCL**、**PORTA**、**PORTB** 和 **W register** 为观察变量。单步运行程序,在适当的时候输入激励,注意观察输入激励后产生的影响。按照下述要求更改程序。

(1) 当按钮按下时,设置端口 B 的位 3 和位 4。

(2) 当按钮按下时,设置端口 B 的其他位。

5.2.2　子例程和栈

当编写较大规模的程序时,就会发现将程序分成多个程序段并将它们放在程序的不同位置,这一方法非常有用。如果在每次使用某个程序段时,都编写一遍该程序段是非常乏味又令人讨厌的,而且代码冗余又浪费存储空间。因此,需要在程序中使用子例程。

子例程是一个结构化的程序段,能在程序的任何位置被调用。一旦子例程执行完毕,主程序继续从子例程之后的位置开始执行。图 5-4 显示了子例程调用所导致的程

序执行过程。在主程序中有一条“调用 SR1”指令。当主程序执行到此处时，跳转到标号所指定的子例程。使用“返回”指令来表示子例程执行完毕并返回到主程序。然后，主程序从“调用”指令之后的指令开始执行。主程序调用 SR1 之后，又调用了另一个子例程 SR2。之后不久，再次调用第一个子例程 SR1。

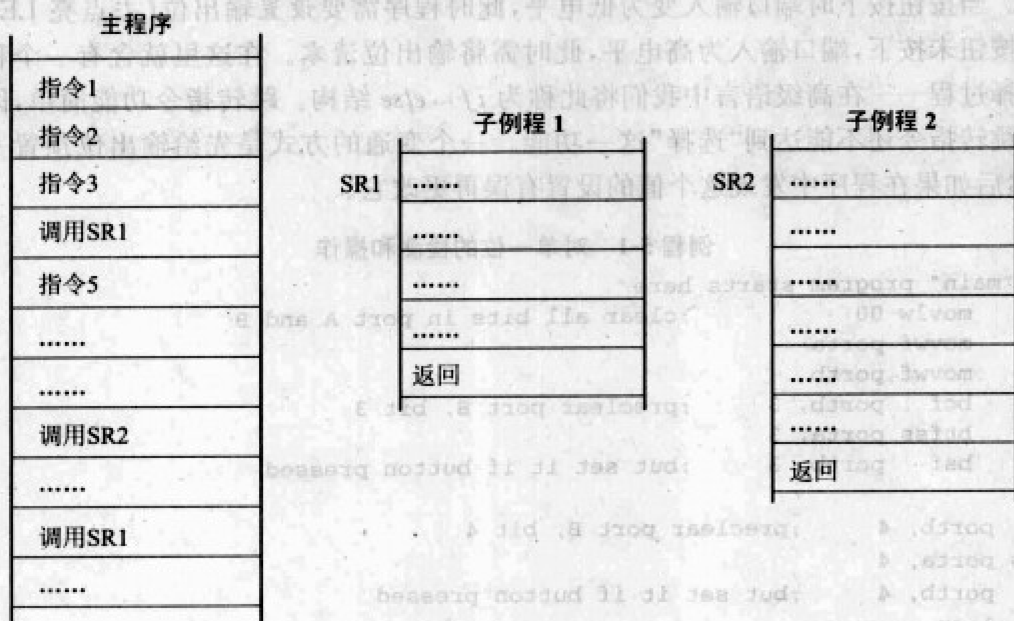


图 5-4　子例程调用

调用(Call)指令的执行包括 2 个方面。首先，它在栈中保存程序计数器的值，以便 CPU 知道程序在子例程执行完后返回到什么地方。然后，加载子例程的起始地址到程序计数器中。这样，程序从子例程处继续执行。返回(Return)指令和调用(Call)指令的作用相反，返回指令将保存在栈顶部的数据(就是调用指令的下一条指令的地址)加载到程序计数器中。然后，程序从该地址继续执行。在程序中，子例程调用指令和返回指令必须总是成对出现。

PIC 16 系列的子例程调用指令和返回指令见附录 1，被简记为 **call** 和 **return**。PIC 16 系列还有一条特殊的返回指令 **retlw**。在后面各节中将学到一些子例程的例子。

在一个子例程中调用一个子例程，称为程序的嵌套调用(nested call)。在子例程中被调用的子例程称为被嵌套调用的子例程。如果遇到这种情况，需要注意的是，每调用一个子例程就会占据一个栈单元，这个栈单元在该子例程返回时被释放。如果我们在一个子例程中调用一个子例程，那么会使用 2 个栈单元；如果再有一个嵌套调用，就会使用 3 个栈单元。由于 16 系列微控制器只有 8 级栈，在子例程调用时必须注意，以免发生“栈溢出(stack overflow)”。

5.3 产生延时和时间间隔

嵌入式系统设计中经常会面临的一个问题是如何处理时间——怎样适时地响应外部事件以及如何计算时间和产生延时。即使我们目前只具备有限的编程知识，但我们可通过开发能产生精确延时的程序循环来处理嵌入式系统中的时序问题。

最初的设想非常简单。将一个存储单元配置为计数器，在该存储单元中存入某个值，然后该值在循环中不断递减，直到变为 0 为止。这个过程花费的时间由计数器的初始值和每次递减循环的时间来决定。

为了实现精确的延时，振荡器的频率要精确而稳定，并且我们需要知道振荡器的频率。因此，我们选择使用晶体振荡器，因为晶体振荡器能够提供非常精确和稳定的频率。使用其他振荡器只能得到粗略的延时，如在电子乒乓球游戏中使用的。对于 PIC，我们应该知道每个指令周期需要 4 个振荡器周期，这从第 2 章的 2.5.1 节可知。

95

例程 5-2 是乒乓球游戏程序中一个简单的延时循环例子。它采用子例程的形式，叫做 **delay5**。进入该子例程后，首先将一个数送至存储单元 **delcntr1** 中。在这里，这个数是 200_D，当然也可设置不同的值以产生不同长度的延时（由于存储单元只有 8 位宽，所以可设置的最大值为 255_D）。实际的延时循环是以标号 **del1** 开始的代码段。2 条 **nop** 指令不做任何事情，只是延时，用于增加执行一次循环的时间。**nop** 指令之后是 **decfsz** 指令，用于递减之前设置的存储单元 **delcntr1** 的值。如果递减的结果为 0，那么程序跳过后面一条指令，执行 **return** 指令。当执行到第 199 次循环时，delcntr1 的值为 2，递减后的结果为 1，不等于 0，所以不会产生跳转，程序返回到 **del1** 处继续执行。

例程 5-2　延时子例程

```
;Delay of 5ms approx. Instruction cycle time is 5us.
delay5  movlw      D'200'        ;200 cycles called,each taking 5x5=25us
        movwf      delcntr1
del1    nop                      ;1 inst. cycle
        nop                      ;1 inst. cycle
        decfsz     delcntr1,1    ;1 inst. cycle, when no skip
        goto del1                ;2 inst. cycles
        return
```

通过循环体中每条指令的执行时间（见附录 1 中指令集），可以很容易地计算出该延时子例程的整个时间。从例程 5-2 中的注释可知每条指令的执行时间。当 **delcntr1** 从初始值开始倒计时时，循环体由 2 条 **nop** 指令、1 条 **decfsz** 指令和 1 条 **goto** 指令组成。当 **decfsz** 指令不跳转时，它只需 1 个指令周期，而 **goto** 指令总是需要 2 个指令周期。所以，每次循环需要 5 个指令周期。电子乒乓球游戏程序的时钟频率大约为 800kHz，因此一个指令周期的频率为 200kHz，也就是一个指令周期的时间为 5μs。所以，包含这 5 个指令周期的每次循环需要 25μs，调用 200 次循环所花时间为 5ms。

为了实现精确的延时，也有必要考虑最后一次循环的延时时间和子例程进入和退出的时间。在例程 5-2 的最后一次循环中，**decfsz** 产生一个跳转，因此需要 2 个指令周期，但是没有执行 **goto** 指令。

对于产生相对较短时间的延时，采用例程 5-1 中简单的延时循环是很有用的，它产生的延时只有数十毫米。但是，在很多情况下，我们需要较长时间的延时。增加延时的一个简单方法是创建一个与第一个延时程序类似的子例程，在该子例程的循环体中调用第一个延时程序，如例程 5-3 所示。例程 5-3 中的循环调用子例程 **delay5** 100_D次。所以最后的延时大约是 500ms。得到较长延时的另一种方法是在单一的子例程中采用“循环中再包含一个循环”的方法，见例程 5-4。

96

例程 5-3 使用嵌套子例程的方法来产生较长的延时

```
;500ms delay (approx)       ;100 calls to delay5
delay500 movlw      D'100'
         movwf      delcntr2
del2     call       delay5
         decfsz     delcntr2,1
         goto       del2
         return
```

延时程序在嵌入式设计中非常有用，它的应用也十分广泛。但是，在使用延时程序时需要很小心，因为在延时程序运行时，CPU 不能做任何事情的。这就好比要爱因斯坦坐下来数豆子——延时程序不能很好地利用功能强大的资源。在第 6 章中，我们将学习产生延时的其他方法。

5.4 数据处理

我们已经知道，不论数据存储器是 SFR 还是指定保存特殊变量的存储单元，从数据存储器读取或写入单个字节的数据都是很容易的。在之前的例程中都是采用如下指令来存储单一字节的数据：

```
movlw   D'100'
movwf   delcntr2
```

在使用这 2 条指令时，需要指定要保存数据的存储单元的实际地址。在上面这个例子中，**delcntr2** 是一个标号，清晰地指明一个存储单元地址。但是，如果想要处理多个数据块，仍使用这种简单的数据移动方式，程序的性能就会很差。例如，如果要存储一列数据，采用这种简单的方法就不得不需要给每个存储单元指定一个固定的地址，这会带来极大的不便。这时我们可以采用一个指针来指向现在要存储数据的存储单元，它可以改变并指向下一个连续的存储单元。在数据存储器中使用这种方式来读取或写入数据很有用。使用这种方式，可以在程序中使用数据块，对数据块进行操作。这在程序存储器

中也很有用。在这种方式下，可以访问预先设定好的程序数据，而不是去修改它[①]。本节将介绍两种对数据存储器和程序存储器中的数据块进行处理的技术。

5.4.1 直接寻址和文件选择寄存器

图 2-2 和图 2-5 中有一种寄存器叫作文件选择寄存器(File Select Register，FSR)。从图 2-2 可知，数据存储器的地址除了可以嵌入在指令字中外，还可以保存在 FSR 中。保存在 FSR 中的数据存储器地址叫作*间接地址*。一旦指令字中的地址信息指向存储单元 **INDF**[②]，就表示使用 FSR。INDF 并不是一个实际存在的寄存器，它是一个假想的存储单元，使用它仅仅是告诉 CPU 现在采用间接寻址模式来寻址。间接寻址模式的好处在于可以把 FSR 当作一个常规的存储单元来操作，使它指向数据存储器中任何要指向的位置。当寻址 **INDF** 时，使用的指令将对 FSR 指向的存储单元进行操作。

例程 5-7 就是一个使用间接寻址的例子。在例程 5-7 中，我们设计了一个产生一列数据(斐波那契序列)的程序。有关例程 5-7 的详细讨论见 5.6.4 节。

97

5.4.2 查找表

通过 **movlw** 指令我们可以在程序中加入一个字节的常量数据，然后以任意方式使用这个常量数据，在前面的例程中我们已经看到了这一点。在指令代码中包含一个字节的常数对于单字节或较少字节操作是比较好的。但是，假如我们想要在程序中放入一整列在程序执行时会用到的数据，这些数据用于程序执行中，可能是产生波形也可能是在屏幕上生产输出图象，那该怎么做呢？假如我们想对这列数据中曾访问过的那些做上标记以记住它们，又该怎么做呢？此时，**movlw** 指令已经不能胜任这些工作了，我们采用一种方法来配置这列数据，这种方法称为*查找表*(look-up table)。

查找表是保存在程序存储器中的一个数据块，可被程序访问并在程序执行时使用。在冯·诺依曼结构(如图 1-7a 所示)中，建立和使用查找表是很容易的，这是因为冯·诺依曼结构的单一地址和数据总线使得所有存储单元的大小相同并且都很容易访问。在哈佛结构(如图 1-7b 所示)中，建立和使用查找表要困难一些，因为数据必须在独立的存储器映射之间移动，而且存储单元的大小也不同(通常来说，程序存储器和数据存储器的存储单元大小是不同的)。所以，在哈佛结构中(PIC 的存储器组织结构就是采用哈佛结构的)使用一种特殊的技术来创建查找表，通过这种技术我们可以学到一些重要的新思想。

---

① 通常在嵌入式系统中，程序存储器是采用 Flash 实现的，在程序执行时，只可读不能写。——译者注

② 从图 4-13 的 PIC 指令格式中可以看到，指令字的低 7 位是作为文件寄存器地址的。当这 7 位不全为 0 时，表示实际的存储单元地址，此时 CPU 访问这个地址，这就是直接寻址。而当这 7 位全为 0 时，它不表示任何实际的存储单元地址，而仅仅是一个标记，用于告诉 CPU 应当访问 FSR 指向的存储单元，此时 CPU 就是间接寻址了。为了和直接寻址统一，称 7 位 0 指定的是 INDF 存储单元。INDF 仅仅是一个假想的寄存器，实际并不存在。——译者注

图 5-5 显示的是在 PIC 16 系列中实现查找表的方法。该查找表以子例程的形式存在。表中每个字节数据都有一个特殊指令：**retlw**。**retlw** 指令是一个"返回"指令，与前面介绍的"返回"指令（**return**）不同的是它需要一个 8 位立即数作为操作数。当它让子例程返回时，同时把操作数也送入 W 寄存器中。PIC 16 系列中的查找表必须是一列 **retlw** 指令，每条 **retlw** 指令都有一个字节数据。

98

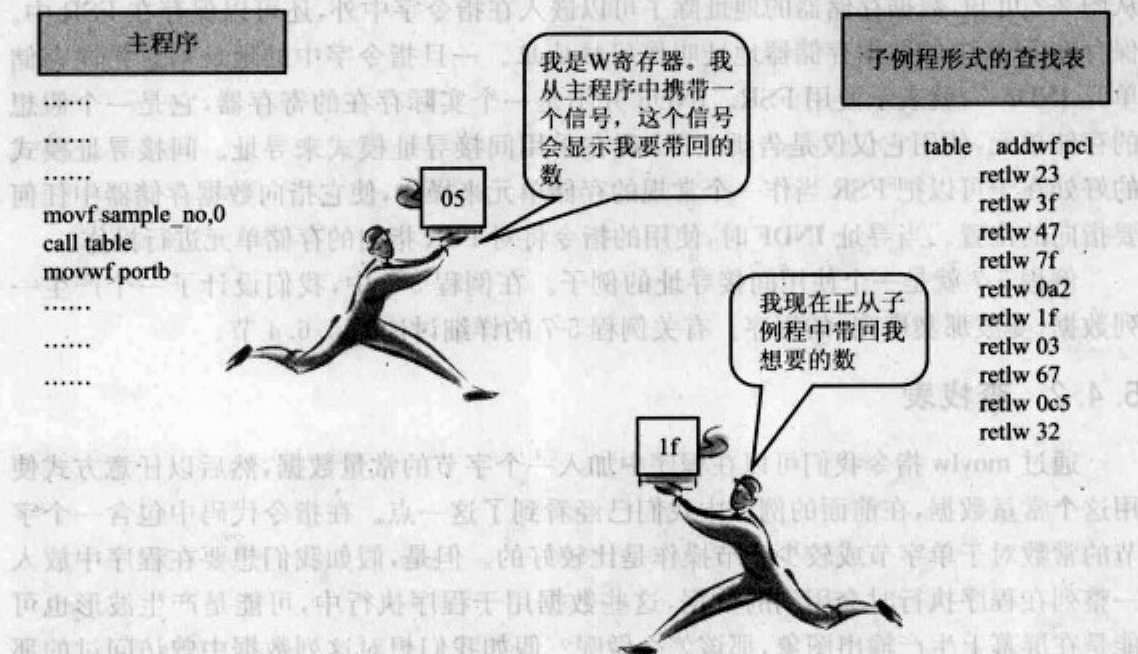


图 5-5 从查找表中获取数据

我们现在需要一种只从查找表的 **retlw** 指令列表中选择其中一条的技术——我们称之为计算 goto(computed go to)。注意查找表子例程的第一条指令：**addwf pcl**。该指令使 **pcl** 加上 **W** 寄存器的值，**pcl** 是程序计数器的低字节。这条指令很容易产生严重的程序错误，因为它直接修改了程序计数器的值，所以在使用它的时候要非常小心。这条指令产生的结果是，一旦程序计数器加上一个数，程序将会向前跳这个数的这么多步后再继续执行。在查找表中，CPU 会执行它跳转到的 **retlw** 指令，然后返回到主程序。值得注意的是，不管查找表有多长，在一次调用中，子例程中仅有 2 条指令被执行。这 2 条指令是 **addwf pcl** 和被跳转到的 **retlw** 指令。对于程序员来说，要确保当子例程调用时，在 W 寄存器中已加载上所需的偏移量①。

我们通过图 5-5 的例子来看看上述过程是怎样实现的。通过 **movf** 指令，主程序将叫做 **sample_no** 的存储单元的值送到 **W** 寄存器中。然后调用子例程 **table**。在该例子中假设存储单元 **sample_no** 保存的值是 5，这正是进入子例程时 W 寄存器保存的值。

① 偏移量即程序要跳转的距离。——译者注

当子例程开始执行时，数值 5 加到 **pcl** 上。因而程序向前跳转 5 步后再执行，也就是执行指令 **retlw 1f**。**retlw** 指令的执行导致子例程返回，同时数值 **1f** 送到 W 寄存器中。主程序就可立即使用这个数，在图 5-5 的例子中，将该数传送到端口 B。

总的来说，W 寄存器就像一个给子例程送信的信使。在进入子例程时，W 寄存器携带着一个给子例程的信号（该信号如同一个指针），通过这个信号告诉子例程它想要表中哪个数，并在子例程返回时带回这个数。

采用这种方法可能存在一个问题——由于只能对程序计数器的低字节操作，所以我们只能访问程序存储器的前 256 个字，或者只能访问一页内的数据。如果查找表非常长，或者数据存放跨页，那么采用"计算 goto"这一方法就会有问题了。在这种情况下，必须完善"计算 goto"这一技术（见参考文献 5.1）。

5.4.3 包含延时循环和查找表的程序例子

例程 5-4 是用于电子乒乓球游戏硬件的一个简单程序，包含了延时循环和查找表。从查找表中获得 8 位数据，并传送给电子乒乓球的 LED，每次数据传送之间有一段延时。程序的整个效果是随机闪烁的 LED。开始的一段程序和例程 4-2 非常相似，但是在这里我们需要给 **pcl** 指定图 2-5 中所示的地址。还需要指定一个 RAM 存储单元，它称为 **pointer**，它的可用地址范围是 0C 到 4F（见图 2-5）。

从标号 **loop** 开始的程序是例程 5-4 的核心程序。正如图 5-5 中的程序一样，**pointer** 的值送至 W 寄存器，并调用一个叫做 **table** 的子例程。子例程返回后，保存在 W 寄存器中的值传送给端口 B，点亮某个 LED，然后调用一个延时循环 **delay**。注意延时循环 **delay** 的结构，**delay** 和例程 5-3 的效果一样，但是只使用了一个子例程，它是采用"循环中再包含一个循环"的方法来增加延时的。程序不停地循环执行，同时检测 **pointer** 是否已经达到最大值。如果达到最大值，将 **pointer** 清零后程序再继续执行。

例程 5-4 使用查找表和延时循环的程序

```
;***********************************************************
;FLASHING LEDs!
;This program continuously outputs a series of led patterns,
;using simulation or ping-pong hardware.
;TJW 5.3.05.                 Tested in simulation 11.3.05.
;***********************************************************
;Clock is 800kHz
;Configuration Word: WDT off, power-up timer on,
;                        code protect off, RC oscillator
;
        list p=16F84A
;
;specify SFRs
pcl      equ   02
status   equ   03
porta    equ   05
trisa    equ   05
portb    equ   06
trisb    equ   06
;
```

```
pointer equ  10
delcntr1 equ 11
delcntr2 equ 12
;
         org   00
;Initialise
start  bsf   status,5        ;select memory bank 1
       movlw B'00011000'
       movwf trisa           ;port A according to above pattern
       movlw 00
       movwf trisb           ;all port B bits output
       bcf   status,5        ;select bank 0
;
;The "main" program starts here
       movlw 00              ;clear all bits in port A
       movwf porta
       movwf pointer         ;also clear pointer
loop   movf  pointer,0       ;move pointer to W register
       call  table
       movwf portb           ;move W register, updated from table SR, to port B
       call  delay
       incf  pointer,1
       btfsc pointer,3       ;test if pointer has incremented to 8
       clrf  pointer         ;if it has, clear pointer to start over
       goto  loop
;
;*****************************************************************
;Subroutines
;*****************************************************************
;Introduces delay of 500ms approx, for 800kHz clock
delay movlw D'100'
      movwf delcntr2
outer movlw D'200'
      movwf delcntr1
inner nop
      nop
      decfsz delcntr1,1
      goto   inner
      decfsz delcntr2,1
      goto   outer
      return

;Holds Lookup Table
table addwf pcl
      retlw 23
      retlw 3f
      retlw 47
      retlw 7f
      retlw 0a2
      retlw 1f
      retlw 03
      retlw 67

      end
```

编程练习 5-2

人们通常认为 PIC 查找表的概念很难理解。所以为了加深对 PIC 查找表的理解，实际仿真一个使用查找表的程序例子是一个不错的主意。创建一个叫 Flashing LED 的项目，复制本书附属资源中例程 5-4 的源代码到你的项目中，然后开始仿真。打开 Watch 窗口，加入观察变量 **PCL**、**PORTB**、**WREG** 和 **pointer**。单步运行程序，仔细观察当子例程进入和退出时 W 寄存器值的变化。使用 Step Over 跳过延时子例程。通过该练习，确保你了解程序的各个阶段和使用的指令。

5.5 逻辑指令

到目前为止，我们已经学习了一些 16 系列的指令，这些指令是 16 系列指令中具有代表性的、经常使用的指令。我们还需要学习一些逻辑指令，这些指令，例如 **andwf**、**andlw**、**iorwf** 或 **xorwf**，在 W 寄存器的值和立即数之间，或在 W 寄存器的值和存储单元值之间执行逻辑操作。这些指令都是基于位的操作。例如，如果使用 **andlw k** 指令，那么立即数 **k** 的位 0 和 W 寄存器值的位 0 相与，立即数 **k** 的位 1 和 W 寄存器值的位 1 相与，以此类推。对于实际的逻辑操作，这些指令非常有用。通常，**and** 指令用于屏蔽一个字中不必要的位，而 **or** 指令用于对一个字中的个别位进行设置。

100～101

作为介绍逻辑指令的例子，我们首先看一看重置例程 5-4 中 pointer 的另外一种方法。在例程 5-4 中，每当 pointer 递增到 8(0000 1000_B)时，它就需要重新置为 0。pointer 较高的 5 位没有用到，那么为什么不通过屏蔽 pointer 的较高 5 位的方法来代替例程 5-4 中 pointer 清零的方法呢？

例程 5-5 是使用逻辑指令 **andlw** 来实现 pointer 清零的另一种方法。在例程 5-5 中，将 pointer 的值与数 07(即 0000 0111_B)相与，以此来代替在每次 pointer 递增时都检测 pointer 的值。当 pointer 递增到 0000 1000_B时，位 3(值为 1)与 07 的位 3 相与变为 0，同时 pointer 的值变为 0。

例程 5-5 逻辑指令介绍

```
loop   movf   pointer,0        ;move pointer to W register
call table
movwf portb       ;move W register, updated from table SR, to port B
call delay
incf pointer,0      ;increment pointer, place result in W reg
andlw 07
movwf pointer
goto loop
```

5.6 算术指令和进位标志

16 系列指令集中有 6 条算术指令：**addwf**、**addlw**、**subwf**、**sublw**、**incf** 和 **decf**。这 6 条

指令对微控制器进行算术运算非常重要。在后面的例程中我们会发现这一点。

5.6.1 加法指令

加法指令的用法简单易懂。可以通过 **addwf** 指令使 W 寄存器的值加上所指定的存储单元的值,也可以通过 **addlw** 指令使 W 寄存器的值加上一个立即数。当两个 8 位数据相加得到一个 9 位的数时,处理过程会稍微复杂一点。结果的第 9 位就是状态寄存器中的进位标志(Carry flag)。

5.6.2 减法指令

减法指令与加法指令类似。除了极性相反外,进位位的作用就如同一个借位(见图 2-3,状态寄存器)。所以,如果减法的结果为正,那么进位位置位。如果结果为负,那么进位位清零。

5.6.3 一个算术程序例子

例程 5-6 是一个用算术指令做简单算术操作的例子。正如例程 5-6 前面的注释信息所描述的,该程序通过连续的加法操作来产生斐波那契数列。在程序中有一个计数器,用于表示已经产生的斐波那契数的个数。当计数器递增,产生的斐波那契数超过 8 位的范围(即溢出)时,计数器开始递减,通过连续的减法来产生反转的斐波那契数列。在每条加法指令后检测进位位来判断计数器值是否溢出(超过 8 位的范围)。

程序开始时,首先将斐波那契数列的前 3 个数存放在存储器中。然后从标号 **forward** 开始执行,对数列的最近的 2 个数进行操作。将数列的最近的 2 个数相加,同时检测进位位。如果进位位没有被置位,那么计数器递增,程序将目前生成的斐波那契数列最新的前 3 个数存入 fib0、fib1、fib2 中。程序返回到 **forward** 继续这个循环。如果进位位被置位,即最近 2 个斐波那契数的和超出了 8 位的范围,此时程序跳转到标号 **reverse** 的位置开始执行,通过减法指令来产生反转的斐波那契数列。程序检测计数器的值是否为 3,来决定是否跳转回 **forward**。

例程 5-6 生成斐波那契数列

```
;*************************************************************************
;In a Fibonacci series each number is the sum of the two previous
;ones, e.g. 0,1,1,2,3,5,8,13,21.....
;This program calculates Fibonacci numbers within an 8-bit range,
;first going up and then down.
;Program intended for simulation only, hence no input/output.
;The program demonstrates addition, subtraction, compare.
;TJW 17.3.05.                              Tested by simulation 18.3.05
;*************************************************************************

   list p=16F84A
;no i/o ports used
status equ 03
```

```
c      equ  0
z      equ  2
;these memory locations hold the three highest values of the Fibonacci series
fib0   equ 10          ;lowest number (oldest when going up,
                              ;newest when reversing down)
fib1   equ 11          ;middle number
fib2   equ 12          ;highest number
fibtemp equ 13         ;temporary location for newest number
counter equ 14         ;indicates value reached, opening value is 3
   org 00
;preload initial values
       movlw 0
       movwf fib0
       movlw 1
       movwf fib1
       movwf fib2
       movlw 3
       movwf counter ;we have preloaded the first three numbers,
                            ;so start count at 3
;
forward movf  fib1,0
       addwf fib2,0
       btfsc status,c       ;test if we have overflowed 8-bit range
       goto  reverse        ;here if we have overflowed, hence reverse down
       movwf fibtemp        ;latest number now placed in fibtemp
       incf  counter,1
;now shuffle numbers held, discarding the oldest
       movf  fib1,0         ;first move middle number, to overwrite oldest
       movwf fib0
       movf  fib2,0
       movwf fib1
       movf  fibtemp,0
       movwf fib2
       goto  forward
;when reversing down, subtract fib0 from fib1 to form new fib0
reverse movf fib0,0
       subwf fib1,0
       movwf fibtemp        ;latest number now placed in fibtemp
       decf  counter,1
;now shuffle numbers held, discarding the oldest
       movf  fib1,0         ;first move middle number, to overwrite oldest
       movwf fib2
       movf  fib0,0
       movwf fib1
       movf  fibtemp,0
       movwf fib0
;test if counter has reached 3, in which case return to forward
       movf  counter,0
       sublw 3
```

```
    btfsc status,z
    goto  forward
    goto  reverse

  end
```

编程练习 5-3

在 MPLAB® 中创建一个叫 Fibonacci 的项目。从本书附属资源中复制例程 5-6 的源文件到该项目中并仿真。在 Watch 窗口中显示 **counter**、**fib0**、**fib1**、**fib2**、**fibtemp**、**WREG** 和 **STATUS**。单步运行程序，并通过 **fib0**、**fib1** 和 **fib2** 观察斐波那契数列的产生过程。在 8 位的范围内，一共可以产生多少个斐波那契数？当数值超过 8 位的范围时，进位位被置位且程序开始产生反转的斐波那契数列，通过 Watch 窗口观察这一过程。需要注意的是，在每条减法指令之后，进位位都被置位(进位位现在的作用就如同$\overline{借}$位)。在 **reverse** 处暂停程序，给 **fib0** 和 **fib1** 强制赋值，使得减法的结果为负。单步运行程序到减法指令，观察 W 寄存器的结果，并且会发现进位位清零了。注意最后一段程序，懂得 **counter** 的值是如何和立即数 3 比较的以及程序是如何返回到 **forward** 的。

5.6.4 通过间接寻址来保存斐波那契数列

例程 5-7 扩展了斐波那契数列程序(如例程 5-6 所示)，通过间接寻址将产生的斐波那契数列保存在存储器中，见程序的主要代码段。例程 5-7 增加的代码用粗体标明。[104] 存储在数据存储器中的数据块的起始地址是 20_H。采用传统的寻址方式将数列的前 3 个数存入数据存储器中(当然也可以采用间接寻址的方式，但代码行数会稍微多一些)。然后，将 23_H 送至 FSR 中，23_H 是数据块的下一个单元的地址。然后，程序进入主循环。在主循环中，每新产生一个斐波那契数，就通过 **movwf indf** 指令将它保存在数[105] 据块中，然后 FSR 的值递增，直到它达到预先设定的最大值。

例程 5-7 通过间接寻址来保存斐波那契数列

```
;...
(opening lines of program omitted)
;...
;these memory locations hold the most recent numbers in the Fibonacci series
fib0    equ 10  ;lowest number(oldest when going up; newest when reversing down)
fib1    equ 11  ;middle number
fib2    equ 12  ;highest number
fibtemp equ 13  ;temporary location for newest number
counter equ 14  ;indicates which value we have reached, opening value is 2

    org 00
;preload initial values
```

```
        movlw 0
        movwf fib0
        movwf 20          ;save at start of list
        movlw 1
        movwf fib1
        movwf 21          ;save in list
        movwf fib2
        movwf 22          ;save in list
        movlw 3
        movwf counter ;have preloaded the first three numbers, so start at 3
        movlw 23          ;Initialise File Select Register, with next location for
        movwf fsr            ;list of numbers to be saved
;
forward movf fib1,0
        addwf fib2,0
        btfsc status,c ;test if we have overflowed 8-bit range
        goto  reverse ;here if we have overflowed, hence reverse down the series
        movwf fibtemp ;latest number now placed in fibtemp
        movwf indf        ;save by indirect addressing
        movf  fsr,0       ;test to see if FSR is at top of range
        sublw 30
        btfss status,z
        incf  fsr,1       ;increment FSR, if available range not full
        incf  counter,1
;now shuffle numbers held, discarding the oldest
          movf  fib1,0  ;first move middle number, to overwrite oldest
          movwf fib0
          movf  fib2,0
          movwf fib1
          movf  fibtemp,0
          movwf fib2
          goto  forward
```

编程练习 5-4

在 MPLAB 中创建一个叫 Fibo+storage 的项目，也可以自己选择一个项目名称。从本书附属资源中复制例程 5-7 的源文件到该项目中并仿真。选择 View>File Registers 来打开 File Registers 窗口。使用“Step-info”来单步运行程序。在 File Registers 窗口中观察在以 20_H 为起始位置的数据存储块中存放的斐波那契数列。程序执行完成时，在你的项目中应该有一个和图 5-6 类似的窗口。从图 5-6 中可以看到 SFR 的 2 个存储区[①]，并留意 **PCL、STATUS** 和 **FSR** 在 2 个存储区中的位置。当 **FSR** 达到最大值时，程序会做些什么？通过改变 **sublw 30** 指令中立即数的值来重新定义数据块的大小。然后观察程序行为会有何变化。

---

① 区 0 的地址 00h～7Fh；区 1 的地址 80h～FFh。——译者注

```
File Registers
Address 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F          ASCII
0000    -- 00 15 19 2F 00 00 00 -- -- 00 00 00 00 4A 00 .../...  --....J
0010    10 69 79 E2 0F 00 00 00 00 00 00 00 00 00 00 00 iy.....  .......
0020    00 01 01 02 03 05 08 0D 15 22 37 59 90 69 79 E2 .......  ."7Y.iy
0030    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .......  .......
0040    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .......  .......
0050    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .......  .......
0060    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .......  .......
0070    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .......  .......
0080    -- FF 15 19 2F 3F FF FF -- -- 00 00 00 00 00 -- .../?..  --.....
0090    -- 00 FF 00 00 -- -- -- 02 00 -- -- 07 00 00 00 ....---  .--....
00A0    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .......  .......
Hex  Symbolic
```

图 5-6　在起始地址为 20_H 的数据块中保存的斐波那契数列

5.7　更复杂的汇编程序

即使对于一个简单的汇编程序，也可以编写得非常复杂，但我们需要程序能尽量简短易懂。本节介绍了一些简化和优化程序的方法。

5.7.1　包含文件

106

通过汇编伪指令 **#include** 可以在汇编程序中包含任何文件，从而避免将所有的代码都粘贴在一起。目前在应用中需要开发的汇编程序越来越复杂，规模也越来越庞大。如果将所有代码都粘贴在一起组成一个很大的汇编程序，这会给程序的开发和管理带来麻烦。采用 **#include** 伪指令就可以避免这种情况。在程序中使用 **#include** 伪指令包含的文件叫做包含文件(Include File)。刚开始学习时，我们使用 **#include** 伪指令最多的情况是，通过包含文件来代替在汇编程序起始处直接定义所有的微控制器指定的存储单元。和其他汇编器一样，MPLAB 给每个微控制器都提供了包含文件，在包含文件中使用 **equ** 语句对所有的 SFR 和 SFR 的所有位都做了定义。在 MPLAB 的其中一个文件夹下可找到这些包含文件。

对于像 16F84A 这样的小型微控制器，使用包含文件是很有用的，它的包含文件只有数页长。对于较大的微控制器，使用包含文件几乎是必须的了，因为较大的微控制器有巨大的 SFR 阵列，因而也有相当长的包含文件。一旦使用包含文件，就必须确保在程序中引用的微控制器指定的标号和包含文件中的相同。通过例程 5-8，我们可以看到即使对于一个非常小的应用来说，使用包含文件也有很多好处。

例程 5-8　使用包含文件

```
;specify SFRs
timer   equ     01
status  equ     03
```

```
porta    equ    05                 #include p16f84A.inc
trisa    equ    05                 OR
portb    equ    06                 include p16f84A.inc
trisb    equ    06
intcon   equ    0B
```

左边的代码（微控制器指定的标号还有很多，未全部列出）在汇编程序中可用右边的代码代替

5.7.2 宏指令（Macro）

从前面的程序中，我们发现对于 RISC 处理器的汇编程序进行开发十分困难，因为每条独立的指令功能有限。在 4.9 节中，我们知道 CISC 指令集提供了一些功能较为强大的指令，但使用这些指令也只能给汇编语言编程带来一定的好处。那么有没有一种方法，使我们能够在现有的汇编环境下充分发挥指令集的功能呢？

答案是使用宏指令（macro）。宏指令是由程序员定义并指定名称的一组指令。一旦宏指令被定义，就可以在程序中随时使用它们。在某些方面，宏指令的使用比子例程更方便，但是在使用上和子例程有所不同。当汇编器对源代码进行汇编时，会将代码中的宏指令用组成它的原始指令来代替。所以，使用宏指令是一种快速编程的形式，而不是一种使程序结构化的方法。

例程 5-9 是在电子乒乓球游戏程序（附录 2）起始处插入 3 条宏指令后的程序。宏指令本身位于伪指令 **macro** 和 **endm** 之间。已经给宏指令定义了参数，这些参数是供宏指令使用的数据。宏指令 **movlf** 将常数送至存储单元中，它需要两个参数：**const** 和 **address**。采用同样的方式定义宏指令 **bfbset**（如果寄存器文件单元的位 b 置位，程序分支）和 **bfbclr**（如果寄存器文件单元的位 b 清零，程序分支）。使用这 3 条宏指令只需非常少的代码行，每次使用宏指令都能节省一行代码。因此，在 **wait** 循环中原本 8 行的代码变为了 4 行。

例程 5-9　在电子乒乓球游戏程序中使用宏指令

```
;now ready for action
;macro to move a literal value to a file
movlf       macro const,address
            movlw const
            movwf address
            endm
;macro to branch if a specified bit is set
bfbset macro file,bit,target
            btfsc file,bit
            goto  target
            endm
;macro to branch if a specified bit is clear
bfbclr macro file,bit,target
            btfss file,bit
            goto  target
```

```
             endm
wait         movlf   04,porta         ;at rest, "out of play"
             movlf   00,portb         ;all play leds off
;both paddles must initially be clear before play allowed to commence
             bfbclr porta,4,wait      ;go to wait if right paddle pressed
             bfbset porta,3,wait      ;go to wait if left paddle pressed
;
```

编程练习 5-5

用例程 5-9 的代码替换电子乒乓球游戏程序中相应部分的代码。将这些代码进行汇编，然后打开列表文件(.lst 文件)。在列表文件中可以看到，宏指令定义并不占据任何存储空间，注意观察在引用宏指令时，宏指令是如何变为组成它的原始指令的，也就是直接将宏指令由组成它的原始指令代替。在整个程序中，在所有能用这 3 条宏指令的地方都使用了它们。你在多少个地方使用了这 3 条宏指令，同时一共节省了多少行代码呢？在整个程序中，还有没有其他代码也可以定义成宏指令？

5.7.3 MPLAB 特殊指令

Microchip 公司还定义了一组"特殊指令"，缓解了 RISC 指令集的指令功能有限这一问题。这组指令可被汇编器识别，并转换为等效的汇编指令。表 5-1 列举了一些特殊指令。完整的特殊指令见参考文献 4.1 的附录 B.11。大多数特殊指令使用或操作状态寄存器的 Z 位和 C 位。一些指令，例如 **bc** 或 **bnc**，并未节省代码行数，但是能够使编程更加清晰。其他一些指令，例如 **addcf**，具有一些在指令集中本来没有的新的实用功能，它们与 CISC 指令非常相似。

108

表 5-1　一些 MPASM™ 特殊指令

助记符	描　述	等效的汇编指令	受影响的状态标志
addcf f,d	f 加上数字进位	BTFSC 3,1 INCF f,d	Z
bc k	如果有进位，跳转	BTFSC 3,0 GOTO k	—
bnc k	如果没进位，跳转	BTFSS 3,0 GOTO k	—
clrc	进位位清零	BCF 3,0	—
movfw f	F 的内容送至 W	MOVF f,0	Z
subcf f,d	F 减去进位	BTFSC 3,0 DECF f,d	Z
tstf f	检测 f	MOVF f,1	Z

5.8 MPLAB 仿真器的更多用处

从前面的学习中可知，软件仿真器作为运行程序和观察输出的一种方法，有着非常巨大的应用价值。我们目前仅使用了仿真器的一些简单的控制，如单步运行、连续单步运行或连续运行。然而，随着开发的程序越来越复杂，我们需要使用软件仿真器的更复杂的功能来运行程序，并观察程序的行为。

5.8.1 断点

如果程序很长，在仿真时仍单步执行它们就会变得非常麻烦。我们需要一种方法来使程序连续运行，穿过我们不关注的代码，而停在我们需要观察的代码处那里并观察程序在执行什么操作。使用断点(breakpoint)可以实现这一功能。在最简单的断点模式下，断点可使程序运行到指定的指令，然后，程序停止执行，此时可以观察存储器和寄存器的值。在 MPSIM™中，双击程序窗口中的指令就可设置一个断点，采用同样的方法就可取消设定的断点。断点的个数没有限制，所以可在仿真中随意使用。

编程练习 5-6

斐波那契程序可能是目前我们见过的最长的程序了。如果我们想观察程序内部较深位置的程序行为，采用单步运行程序的方式将非常麻烦。打开你之前创建的 Fibonacci 项目(或者在这里新创建该项目，如果之前没有创建该项目)，并通过单击 Debugger>Select Tool>MPLAB SIM 来选择 MPLAB 仿真器。打开.asm 源文件，双击源代码中标有 reverse 的一行代码，这样就会插入一个断点，出现一个断点符号，如图 5-7 所示。可以再次双击该行来取消断点。复位仿真器并运行。注意程序在断点处是怎样停止的。此时，你可以观察所有的窗口，然后以任意方式继续运行程序，例如单步运行。在你的源文件中，试着在图 5-7 中的第 2 个设置断点的代码行上设置一个断点，然后运行程序到这个断点的位置。

109

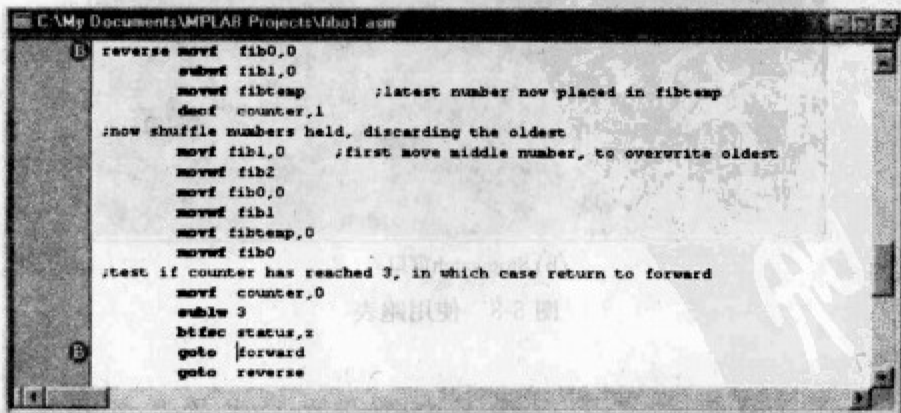
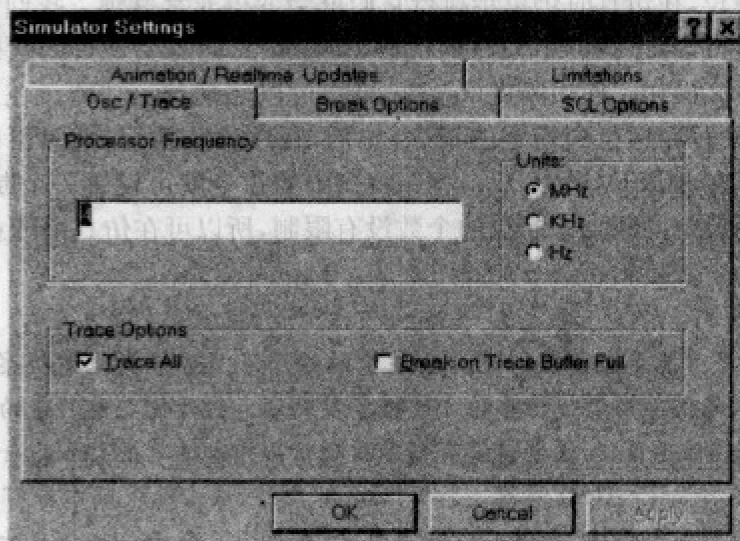


图 5-7 在斐波那契程序中插入断点

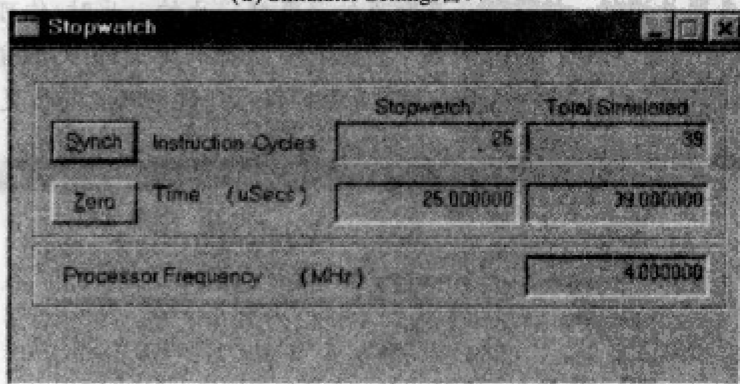
5.8.2 跑表

软件仿真器的一个缺点是不能实时地运行程序，然而在嵌入式系统中我们特别希望能够知道程序的实时行为。仿真器提供了一个跑表(Stopwatch)的功能，它能够仿真精确的时间。要使用跑表的功能，只需让仿真器"知道"振荡器的频率是多少。因为跑表可以记录已执行的指令周期数，所以它能计算出所花的时间。

在 MPSIM 中，单击 Debugger>Settings 打开 Simulator Settings 窗口，选择 Osc/Trace 标签(如图 5-8a 所示)，在该窗口中设置振荡器频率。单击 Debugger>Stopwatch 打开 Stopwatch 窗口(如图 5-8b 所示)。



(a) Simulator Settings窗口



(b) Stopwatch窗口

图 5-8 使用跑表

编程练习 5-7

仍然打开 Fibonacci 项目，将处理器的频率设置为 4MHz。那么指令周期就是 1μs。保留例程 5-6 中设置的断点。单击 Debugger>Stopwatch 打开 Stopwatch 窗口，单击窗

口中的 Zero 按钮，清零跑表。复位仿真器并运行程序到断点处，此时跑表应当显示 166μs。你能解释为什么结果是 166μs 吗？

5.8.3 跟踪

前面使用的 MPSIM 的各个窗口能够告诉我们任一时刻处理器的状态和存储单元的内容，但是不能告诉我们程序执行的历史信息。即使程序在断点处暂停执行了，我们也不能知道程序执行的历史信息，因为很多条程序路径都可使程序运行到这个断点的位置。跟踪(Trace)功能记录最近的程序执行信息。在跟踪存储器(Trace memory)中，仿真器保存所有已执行指令的一条连续的记录，这可以在程序停止时观察到。 [110]

MPLAB 提供跟踪功能，跟踪存储器的大小为 32767 行。单击 Debugger>Settings 打开"Simulator Settings"窗口，选择 Osc/Trace 标签(如图 5-8a 所示)，在该窗口中启用跟踪功能。注意：启用跟踪功能后，仿真器的速度会有所降低。单击 View>Simulator Trace，可以打开 Trace 窗口(如图 5-9 所示)。Trace 窗口标题栏中 SA、SD、DA 和 DD 的意思是(其他标题一目了然)：

SA=源地址(Source address)——源数据的地址或符号 [111]

SD=源数据(Source data)——源数据的值

DA=目标地址(Destination address)——目标数据的地址或符号

DD=目标数据(Destination data)——目标数据的值

编程练习 5-8

回到编程练习 5-6 中。按照上述的方法启用跟踪功能，然后复位仿真器并运行程序到标号 **reverse** 的断点处。现在打开 Trace 窗口，如图 5-9 所示。可以看到，在 Trace 窗口中列出了所有最近执行的指令，最后一条执行的指令是 **goto** 指令(在 142 行)，**goto** 指令使程序运行到该断点行。从 SD 栏和 DD 栏可以看到 Fibonacci 数列是如何一步一步通过指令产生的。状态寄存器的值(在 141 行)是 19_H，也就是 $0001\ 1001_B$。这表明位 0(进位标志)已被置位，相应地，**goto** 指令被执行。

Trace

Line	Addr	Op	Label	Instruction	SA	SD	DA	DD	Cycles
134	000F	0812		MOVF 0x12, W	0012	90	0012	90	0000009B
135	0010	0091		MOVWF 0x11	----	--	0011	90	0000009C
136	0011	0813		MOVF 0x13, W	0013	E9	0013	E9	0000009D
137	0012	0092		MOVWF 0x12	----	--	0012	E9	0000009E
138	0013	2807		GOTO 0x7	----	--	----	--	0000009F
139	0007	0811	forward	MOVF 0x11, W	0011	90	0011	90	000000A1
140	0008	0712		ADDWF 0x12, W	0012	E9	0012	79	000000A2
141	0009	1803		BTFSC 0x3, 0	0003	19	----	--	000000A3
142	000A	2814		GOTO 0x14	----	--	----	--	000000A4

图 5-9 显示了一段斐波那契程序的 Trace 窗口

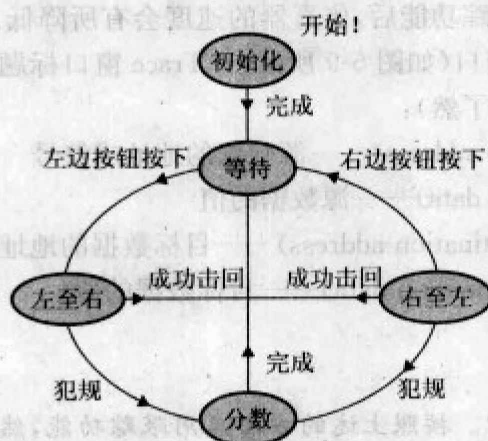
5.9 电子乒乓球游戏程序

现在让我们来看一个完整的电子乒乓球游戏程序(见附录 2),这对于我们进一步掌握前面所学的汇编知识很有帮助。在本节中,我们并不是要逐条阅读别人写的汇编代码(事实上,即使阅读你自己编写的汇编代码通常也是很困难的。)所以如果刚开始你觉得很难,这是正常的,不必担心,因为你还不了解整个程序的功能和结构。

5.9.1 程序结构

112

首先我们从整体上来了解一下游戏程序的结构。整个程序描述了使用状态图其功能,如图 5-10 所示,如果采用流程图来描述会更困难。



犯规	计分
1. 在球还未到达你这边时,就试图击球	当你的对手犯规时,你得分
	成功击回
2. 在球到达你这边时,未击球	在球到达你这边时,击中球

注:当离你最近的LED点亮时,球到达你这边

图 5-10 采用状态图表示的电子乒乓球游戏程序

程序从“*初始化*”状态开始执行。该状态执行完成后,程序立即进入“*等待*”状态。在“*等待*”状态,程序一直等待直到条件满足才进入下一状态。如果左边的选手按下按钮,那么程序进入“*左至右*”状态。在该状态中,“球”从最左边的位置开始,向右边移动。如果有人犯规(在程序中定义了游戏规则)或者“球”在到达最右边时被对方选手成功击回,程序从该状态退出。如果没有选手犯规,程序就在“*左至右*”状态和“*右至左*”状态间来回执行。当一个选手失误未接到“球”(被认为犯规),游戏就进入“*计分*”状态。分数统计完之后,程序离开此状态,返回到“*等待*”状态。

仔细分析电子乒乓球程序的状态图,然后试着在汇编程序中找到每个状态。这 5 个状态中的其中 3 个(*初始化、等待和计分*)应该很容易找到。其实,我们在第 4 章已经

见过了。游戏实际运行中有2个状态(左至右和右至左),这2个状态不太容易理解。这2个状态是对称的,因此如果明白其中一个状态就能马上明白另一个了。

113

程序的整体结构采用状态图来表示是最好的,但实际的“左至右”状态和“右至左”状态肯定是循环结构,因此最简单的表示方式是流程图(见图5-11)。

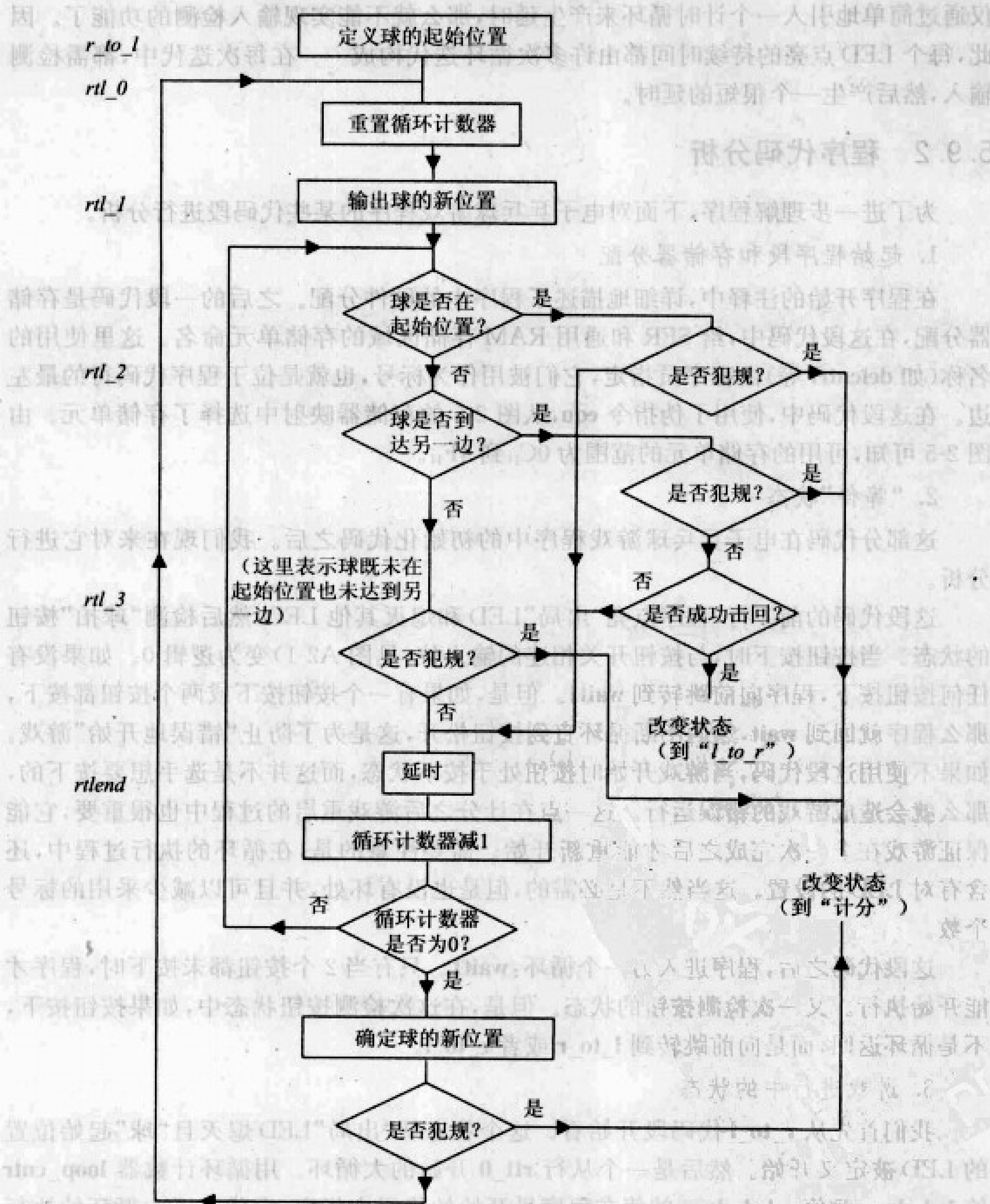


图5-11 “左至右”或“右至左”状态的流程图

114

这可能是我们遇到的第一个详细而复杂的程序，虽然该程序从外观上看很简单。在这个状态中需要设置大量的必要条件。通过点亮一列LED来表示“球”的“移动”，每个LED点亮时必须维持一个设定的时间周期。持续地检测“球拍”按钮的状态，有时按下“球拍”按钮是一个符合规则的动作，有时按下“球拍”按钮则违反了游戏规则。如果仅通过简单地引入一个计时循环来产生延时，那么就不能实现输入检测的功能了。因此，每个LED点亮的持续时间都由许多次循环迭代构成——在每次迭代中，都需检测输入，然后产生一个很短的延时。

5.9.2 程序代码分析

为了进一步理解程序，下面对电子乒乓球游戏程序的某些代码段进行分析。

1. 起始程序段和存储器分配

在程序开始的注释中，详细地描述了程序中的硬件分配。之后的一段代码是存储器分配，在这段代码中，给SFR和通用RAM存储区域的存储单元命名。这里使用的名称(如 **delcntrl** 等)由程序员指定，它们被用作为标号，也就是位于程序代码行的最左边。在这段代码中，使用了伪指令 **equ**，从图2-5的存储器映射中选择了存储单元。由图2-5可知，可用的存储单元的范围为 $0C_H$ 到 $4F_H$。

2. “等待”状态

这部分代码在电子乒乓球游戏程序中的初始化代码之后。我们现在来对它进行分析。

这段代码的前4行，程序点亮“出局”LED和熄灭其他LED，然后检测“球拍”按钮的状态。当按钮按下时，与按钮开关相连的输入位(见图A2-1)变为逻辑0。如果没有任何按钮按下，程序向前跳转到 **wait1**。但是，如果有一个按钮按下或两个按钮都按下，那么程序就回到 **wait**，然后不断循环直到按钮松开，这是为了防止“错误地开始”游戏。如果不使用这段代码，当游戏开始时按钮处于按下状态，而这并不是选手想要按下的，那么就会造成游戏的错误运行。这一点在计分之后游戏重启的过程中也很重要，它能保证游戏在上一次完成之后才能重新开始。需要注意的是，在循环的执行过程中，还含有对LED的设置。这当然不是必需的，但是也没有坏处，并且可以减少采用的标号个数。

这段代码之后，程序进入另一个循环：**wait1**。只有当2个按钮都未按下时，程序才能开始执行。又一次检测按钮的状态。但是，在这次检测按钮状态中，如果按钮按下，不是循环返回，而是向前跳转到 **l_to_r** 或者 **r_to_l**。

3. 游戏进行中的状态

我们首先从 **r_to_l** 代码段开始看。这个状态从“出局”LED熄灭且“球”起始位置的LED被定义开始。然后是一个从行 **rtl_0** 开始的大循环。用循环计数器 **loop_cntr** 给 **led_durn** 赋值。**led_durn** 的值在程序最开始的代码中指定，它代表了内循环的执行次数。内循环从行 **rtl_1** 开始。内循环的主要功能是检测是否有选手犯规，通过当前

球的位置来判断是否犯规。在图 **5-11** 的流程图中给出了 **r_to_l** 代码段的结构，阅读这段代码可以了解犯规是怎样检测的。一旦有人犯规，就进入“计分”状态，统计分数。在内循环 **rtl_1** 的末尾调用了一个延时子例程。然后循环计数器减 1。如果循环计数器值为 0，那么通过移位存储单元 **led_posn**，给球设置一个新的位置。如果球到达另一边，并离开最后一个位置（8 位的存储单元），则开始统计分数，这种情况发生在球已经到达另一边并且没有被成功击回的时候。

“计分”状态由 2 个简单的部分组成。它点亮“分数”LED，再调用半秒的延时程序，然后熄灭“分数”LED。然后，程序离开该状态，返回到“等待”状态。

5.10 电子乒乓球游戏程序仿真

电子乒乓球游戏程序并不是很复杂，但是在程序中使用了大量的循环和延时，因而在仿真时详尽全面地对程序进行测试就成了一个问题。适当地使用仿真器的各种功能可以解决这个问题。一般来说，如果想详细地分析一个程序段，可以采用单步运行或连续单步运行。对于你想快速跳过的代码段，可以在代码段后设置断点，连续运行程序直到断点处。本节的内容能够指导你对本程序进行仿真。

首先要确保你已经创建了电子乒乓球游戏项目，并在项目中包含了电子乒乓球游戏程序。

5.10.1 设置输入激励

电子乒乓球游戏程序有两个数字输入，即 2 位选手的“球拍”按钮。需要仿真这两个输入。单击 Debugger>Stimulus Controller 打开 Stimulus Controller 对话框。在 Pin 下选择 RA3，在 Action 下选 Set High。对 RA4 做同样的操作。然后再为 RA3 和 RA4 配置 2 个激励，这一次在 Action 下选择 Pulse Low。设置脉冲持续时间为 50ms，代表了实际硬件中一次快速地按钮按下又弹起的时间。你能从程序中计算出一位选手按下按钮的最大理论持续时间吗？

5.10.2 设置 Watch 窗口

单击 View>Watch 打开 Watch 窗口。在第一个下拉框中选择 PCL（观察电子乒乓球游戏程序的 PCL 值，能够跟踪程序的执行，知道程序的执行位置）、**PORTA** 和 **PORTB**，单击 Add SFR 按钮添加到 Watch 窗口内。在第二个下拉框中选择 **duration**、**led_posn** 和 **delcntr1**，单击 Add Symbol 按钮添加到 Watch 窗口内。右键单击 Watch 窗口顶部附近的标题栏，会弹出一个菜单，在菜单中提供了更多的选项，例如你可以选择 Binary，采用二进制显示数据。在你关闭 Watch 窗口时，软件会自动保存你的配置。

5.10.3 单步运行

按 F6 复位程序计数器，然后单步运行电子乒乓球游戏程序。你可以看到当受程

116 序指令影响时，寄存器值的变化。变化的寄存器值用红色高亮显示。

如果1位选手或2位选手的"球拍"按钮设置为低(即按下)，那么程序会阻塞在第一个等待循环 **wait**。(通过观察 Watch 窗口中的 Port A 的值，就可知道按钮的状态)。单击 Stimulus Controller 对话框上的 Fire 按钮设置 RA3 和 RA4 这两根线为高①。你可以观察到 Watch 窗口中的变化。

现在单步运行程序到 **wait1** 循环。在这里循环1次或2次后，将设置的脉冲加载给 RA3。那么此时程序退出 **wait1** 循环，运行到 **l_to_r**。此时端口 B 的值变为 01，因为此时已设置了球的起始位置。从这里开始，你可以继续单步运行，也可以单击 Step Over，或进入 **delay5** 子例程。一旦进入 **delay5** 子例程，就可随时单击 Step Out 跳出该延时循环子例程。显然，一直单步执行 **delay5** 子例程是一件非常无聊的事情。但是即使跳过 **delay5** 子例程，还是不能退出大循环，单步执行程序仍然有其局限性。

5.10.4 连续单步运行

再一次按 F6 复位仿真器，并单击 Animate 使程序运行在连续单步运行模式下。根据自己的需要，在 Debugger>Settings>Debugger Animation 中设置运行速度。现在，你就不能使用 Step Over 功能了，当程序运行到延时子例程时，你会发现程序再次阻塞在延时子例程中。此时，从 File Register 窗口可以观察到 **delcntr** 的值在不断递减。

5.10.5 运行

如果选择 Run 来运行程序，此时无法通过 Watch 窗口观察任何变化，因为在程序运行时存储器窗口不再更新。但是，程序仍可接受输入激励。这对在仿真程序中使用断点很有帮助。

5.10.6 断点

首先在 **l_to_r** 标号处设置一个断点。现在复位仿真 CPU，在 Stimulus Controller 对话框中设置 RA3 和 RA4 为高，然后单击 Run 来运行程序。在 RA3 上加载脉冲激励。那么程序会在你设置的断点处暂停。不用复位，在 **ltr_l** 标号处设置另一个断点，再次单击 Run。在 MPLAB 仿真器中，断点的个数没有限制，所以可随意使用。

5.10.7 跑表

在 Debugger>Settings>Osc/Trace 中设置处理器的频率为 800kHz，这是电子乒乓球游戏的额定频率。在 **delay5** 子例程的第一行设置一个断点，程序运行至此。现在单击 Debugger> Stopwatch 打开 Stopwatch 窗口。单击 Zero 按钮将跑表清零。然后

---

① 在 5.10.1 节中给它们配置的激励就是高电平。——译者注

在 **delay5** 子例程的 return 指令处设置一个断点。运行程序到这个断点。看看跑表的值是否和延时子例程 **delay5** 计算的值一样？跑表是精确计算程序执行时间的一个非常有用的工具。

5.10.8 跟踪

启用跟踪，运行程序到一个断点处，然后观察跟踪存储器的值（单击 View>Simulator Trace 打开跟踪存储器）。

5.10.9 调试整个程序

按照表 5-2 中推荐的断点位置，在程序中插入断点。注意：只能在含有指令的行上才能设置断点。按 F6 复位程序计数器，然后运行程序。从一个断点到另一个断点，单步运行程序，并观察表 5-2 中所列出的 Watch 窗口的值。

表 5-2 在电子乒乓球游戏程序调试中推荐的断点位置

断点位置	程序运行到断点处的行为
wait	检测端口 A 的位 3 和位 4 是否为高？单击 Run 运行到此处的循环，观察到"出局"LED 被程序点亮。通过激励产生器设置位 3 和位 4 为高。单击 Run
wait1	此时端口 A 的值应是 0001 1100。通过激励产生器给 RA4 产生低电平的脉冲激励（即 1 位选手开始游戏）；低电平的脉冲持续 50ms。单击 Run
r_to_l	此时如果已按下 RA3 或 RA4，注意端口 A 的位 3 或位 4 是否为低
l_to_r	单击 Run
rtl_1	单步运行这里的几行代码，可以看到 led_posn 被设置了一个新的数值
ltr_1	这个数值代表了球的位置。单击 Run。注意在每次执行循环体时 loop_cntr 递减。然后观察 led_posn 的变化
rtlend ltrend	完成一次内循环之后，将调用一个延时子例程。单击 Run。为了较快地运行，去掉此处的断点
score_left	如果一次"非法"的按钮按下动作导致一次计分，程序运行至此。单击 Run
score_right	如果一次"非法"的按钮按下动作导致一次计分，程序运行至此。单击 Run
delay5	单击 Step Info 进入延时循环。观察 delcntr1 的设置以及之后的递减。单击 Run。为了较快地运行，去掉此处的断点
delay500	如果"分数"LED 点亮，程序运行至此。"分数"LED 将点亮 0.5s。单击 Run

在几次循环后，你就会明白整个程序的行为了。然后试着在仿真器中进行如下操作：

- 做一次"非法"的按钮按下操作，得到一次计分；
- 不断循环，直到球到达另一边并且在一个"合法"的按钮按下操作之后返回。

5.11 其他仿真器介绍——图形化的仿真器

第 4 章和第 5 章介绍了 MPLAB 和其中的仿真器 MPSIM。这些工具的功能非常强大,而且它们都是免费的。如果我们想花钱购买仿真器,那么有哪些选择呢?

有大量的仿真器可供使用,与 MPSIM 简单的基于文本的界面相比,这些仿真器有更好的界面。图 5-12 显示的是一个在参考文献 5.2 中介绍的仿真器。从图中可以看到,仿真器正在对 16F84 微控制器进行仿真。在仿真器中非常清晰地显示出了 W 寄存器、流水化操作的指令、正在执行的指令、栈、状态寄存器、端口以及程序代码。程序可以连续运行,也可以单步执行,程序运行的内部状态实时更新并显示出来。 118

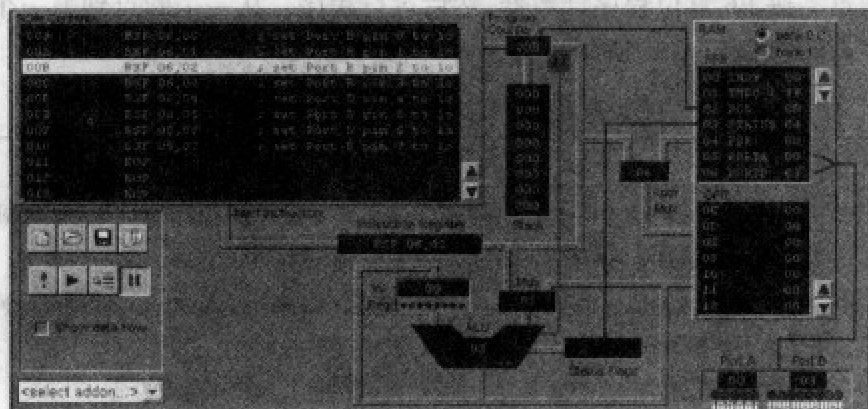


图 5-12 Matrix Multimedia 的 Virtual PICmicro 仿真器

小结

- ☐ 在开发程序时,规划好程序的结构是非常重要的。可以使用流程图和状态图来帮助完成。
- ☐ 在编程中使用大量的技术能够使程序结构清晰、可读性强。这些技术包括子例程、查找表、宏指令和包含文件的使用。
- ☐ 可以使用完整的 16F84A 指令集创建大型的复杂程序。
- ☐ 程序越复杂,在仿真时就需要越专业的仿真技术。

参考文献

5.1. *Implementing a Table Read.* Microchip, Application Note AN556; www.microchip.com.

5.2. Assembly for PICmicro™ Microcontrollers, V3.0. John Becker, Matrix Multimedia Ltd; http://www.matrixmultimedia.co.uk 或 http://www.labvolt.com/ 119

第6章 与计时相关的设备：中断、计数器和定时器

我们日常生活的各个方面都与时间相关，几乎无处不受时间的约束和支配。闹钟叫我们起床，跑表计算所用的时间，定时器用于触发单一事件（例如VCR录像）以及维持周期性的事件（例如房屋加热系统每天在同一时间开始工作）。对于学生和老师来说，根据学校的时间表去上课，这个时间表就是一系列复杂的受时间约束的事件。

对于嵌入式系统来说，时间同样非常重要。在简单的应用层次上，系统也需要适时响应外部事件，还需要计算外部事件之间的时间，并产生基于时间的行为动作。要做到这些，系统主要采用微控制器的2种不同但相关的功能：中断和计数器/定时器。它们都是独立的元件，且两者都非常有用，因此它们广泛地应用于众多微控制器中。几乎每一个微控制器外围设备都会产生中断。从使用脉宽调制的电机控制到串行通信中波特率的产生，所有这些行为动作都由计数器/定时器提供计时功能。在本章中将介绍中断和计数器/定时器的原理，读者需要详细地学习这些原理。由于它们既用在简单的应用中，也用在高级复杂的应用中，在本书中很多地方我们都会提到它们。最终，我们会发现，中断和定时器/计数器都是非常了不起的技术，它们能够加强实时编程。

在本章中，你将学到：

- ☐ 为什么需要中断和计数器/定时器；
- ☐ 基本的中断硬件结构；
- ☐ 16F84A中断结构；
- ☐ 怎样使用中断进行编程；
- ☐ 基本的微控制器计数器/定时器硬件结构；
- ☐ 16F84A Timer 0结构；
- ☐ 计数器/定时器的简单应用；
- ☐ 休眠（Sleep）模式。

如果你愿意，你还将学到：

- ☐ 通过其他微控制器型号的例子来学习中断策略的另一种方法；
- ☐ 深入讨论16F84A中断的问题，特别是它的中断响应延时。

6.1 中断

众所周知，计算机的CPU是一个非常有序的计算实体，它按照顺序一条接一条地

执行程序的指令，以精确和可预测的方式执行被告知的事项。中断却可以打断这一顺序。很容易就能想到，中断的功能是以明确的方式提醒 CPU 有一些重要的外部事件已经发生，停止 CPU 正在做的事，并强制 CPU(以尽可能快的速度)响应发生的外部事件。起初，中断用于当出现紧急外部事件时，例如电源故障、系统过热和子系统主要故障等紧急事件，CPU 能有所感知。中断这一思想被认为非常有效，并且随着时间的推移，越来越多的子系统也都能够产生中断。这导致了中断结构的复杂度增加，同时还需要识别所有不同的中断。

为了能学习和掌握中断，我们既要懂得中断的硬件结构又要掌握使用中断顺利编程的编程技术。从现在开始，将介绍这些内容。

6.1.1 中断结构

不同的微控制器有不同的中断结构。这些中断结构必有多于一个的中断源。通常，一些中断源从微控制器内部产生，另一些中断源来自于微控制器外部。图 6-1 是一个基本的中断结构，说明了中断的主要硬件原理。在左边有一个中断源"中断 X"，它是多个中断源中的一个。如果中断发生，它将设置一个 S-R 双稳态触发器。一旦中断发生，即使只产生一个瞬时的中断，它也会被记录下来。双稳态触发器的输出为被触发器锁存的中断，称为*中断标志*(interrupt flag)。然后，中断标志受到启用信号——*中断 X 启用*(Interrupt X Enable)的门控。如果启用信号为高，那么中断信号传递到一个或门。如果启用信号为低，中断信号就不会向前传播了。如果中断信号启用，它就与微控制器的其他启用的中断信号相或。如果任一个中断输入为高，或门的输出就变为高。或门的输出也受到门控，门控的启用信号是*全局中断启用*(Global Interrupt Enable)。一旦该启用信号为高，前面来的任何中断信号就可到达 CPU 了。当 CPU 响应中断后，需要清零中断标志。在一些处理器中，这由 CPU 自动完成，而在其他一些处理器中，必须由程序来完成。

121

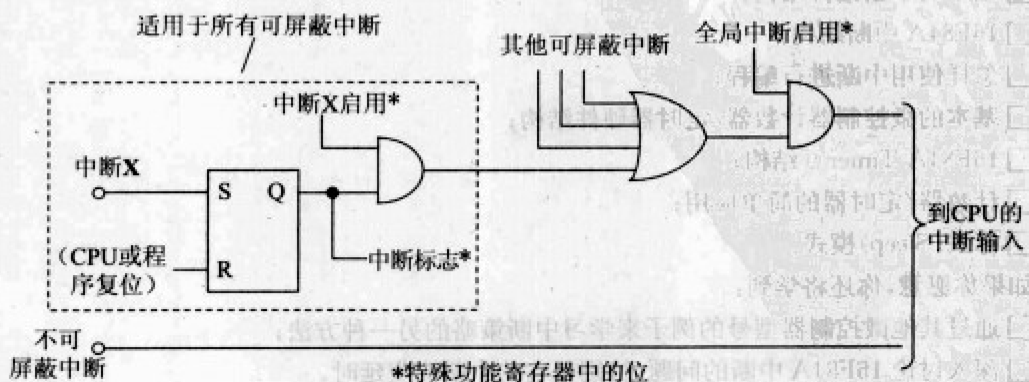


图 6-1 一个简单的基本中断结构

使中断禁止的电路动作有时称为*屏蔽*(masking)。但是，这看起来有些奇怪，中断非常重要，设置中断的目的就是为了向 CPU 报告紧急事件，而中断屏蔽却使微控制器

的中断能力失效。因此,一些微控制器具有不可屏蔽的中断。不可屏蔽的中断总是来自微控制器外部的(也就是说,不会来自片内外围设备),并且被用于连接到最重要的外部中断信号。图 6-1 中也显示了一个不可屏蔽的中断。如果该中断发生,CPU 总会响应,所以没有必要把它存为中断标志,因此对这样的中断,有时在电路中没有将其存为一个中断标志。

6.1.2 16F84A 中断结构

16F84A 有 4 个中断源,这 4 个中断源均可被单独地启用或禁止。

- ❑ 外部中断(External interrupt)。这是唯一的外部中断输入。它和端口 B 的位 0(如图 2-1 所示)共享一个引脚。它是沿触发的。
- ❑ 定时器溢出(Timer overflow)。这是一个由 Timer 0 模块触发的中断,在本章的后半部分将学习这个中断。当定时器的 8 位计数器溢出时,该中断发生。
- ❑ 端口 B 电平变化中断(Port B interrupt on change)。当检测到端口 B 的高 4 位有电平变化时,该中断发生。这种机制已经在 3.4.1 节中学习过。
- ❑ EEPROM 写完成(EEPROM write complete)。当对 EEPROM 存储器的写指令完成时,该中断发生。

16F84A 的中断结构如图 6-2 所示,控制该中断结构的 SFR——**INTCON**,如图 6-3 所示。因为 **INTCON** 寄存器的每一位都出现在图 6-2 中的结构逻辑框图中,所以同时学习图 6-1 和图 6-2 将有助于我们的理解。图 6-2 左边标注的是 16F84A 的 4 个中断源。将图 6-2 与图 6-1 相比,就会发现图 6-2 中没有中断标志触发器。实际的硬件电路中是含有中断标志触发器的,只是在 Microchip 公司的图中没有画出而已。每个中断源都有一个启用线(标有…**E**)和一个标志线(标有…**F**)。因此,线 **T0IF**、**INTF** 等实际上就是中断标志,而不是中断输入本身。除了 EEPROM 写完成标志线,其他所有线都可看作 **INTCON** 中的位。必须注意的是外部中断是沿触发的。该中断响应的沿通过设置 **OPTION** 寄存器的 **INTEDG** 位来控制(见本章后面的图 6-9,因为它主要和 Timer 0 相关)。

122

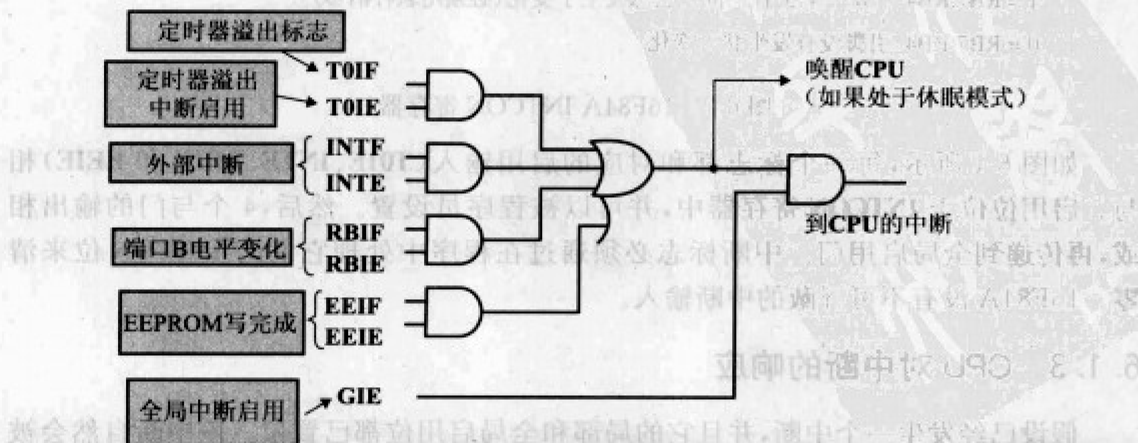


图 6-2 16F84A 中断结构(阴影框中所附标签为作者所加)

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE	EEIE	TOIE	INTE	RBIE	TOIF	INTF	RBIF
位7							位0

位7 **GIE**：全局中断启用位

1 = 启用所有未屏蔽的中断

0 = 禁止所有中断

位6 **EEIE**：EE写完成中断启用位

1 = 启用EE写完成中断

0 = 禁止EE写完成中断

位5 **T0IE**：TMR0 溢出中断启用位

1 = 启用TMR0 溢出中断

0 = 禁止TMR0 溢出中断

位4 **INTE**：RB0/INT外部中断启用位

1 = 启用RB0/INT外部中断

0 = 禁止RB0/INT外部中断

位3 **RBIE**：RB端口电平变化中断启用位

1 = 启用RB端口电平变化中断

0 = 禁止RB端口电平变化中断

位2 **T0IF**：TMR0溢出中断标志位

1 = TMR0 寄存器已经溢出(必须用软件清零)

0 = TMR0 寄存器尚未发生溢出

位1 **INTF**：RB0/INT外部中断标志位

1 = 发生RB0/INT外部中断(必须用软件清零)

0 = 未发生RB0/INT外部中断

位0 **RBIF**：RB端口电平变化中断标志位

1 = RB7:RB4 引脚中至少有一位的状态发生了变化(必须用软件清零)

0 = RB7:RB4 引脚没有发生状态变化

图 6-3 16F84A INTCON 寄存器

如图 6-1 所示，每一个标志都和对应的启用输入(**T0IE**、**INTE**、**RBIE** 和 **EEIE**)相与。启用位位于 **INTCON** 寄存器中，并可以被程序员设置。然后，4 个与门的输出相或，再传递到全局启用门。中断标志必须通过在程序中处理它们的 **INTCON** 位来清零。16F84A 没有不可屏蔽的中断输入。

6.1.3 CPU 对中断的响应

假设已经发生一个中断，并且它的局部和全局启用位都已置位。该中断自然会被 CPU 检测到，并且 CPU 执行一个叫做中断服务程序(Interrupt Service Routine ISR)的

特殊程序段。了解 ISR 如何工作是非常重要的,图 6-4 显示的是 ISR 的流程图。CPU 完成它当前执行的指令,并在栈顶部保存程序计数器的值。这样,CPU 会“知道”在 ISR 完成时要返回的程序位置。为了避免其他可能的中断打断该中断,它也要清零全局中断启用位。

在 PIC® 16 系列中,ISR 必须从中断向量(地址为 0004 的程序存储器单元,见图 2-4)处开始。当中断发生时,地址 0004 加载到程序计数器中,然后程序从复位向量开始执行。无论在哪种处理器中,ISR 都必须以一个特殊的“中断返回”指令结束。在 16 系列中,这条指令就是 **retfie** 指令。当它被检测到时,CPU 设置 **GIE** 为 1,将栈顶部的值加载到程序计数器中,然后重新开始程序执行。这样,程序返回到中断被检测到时的指令的后一条指令。

124

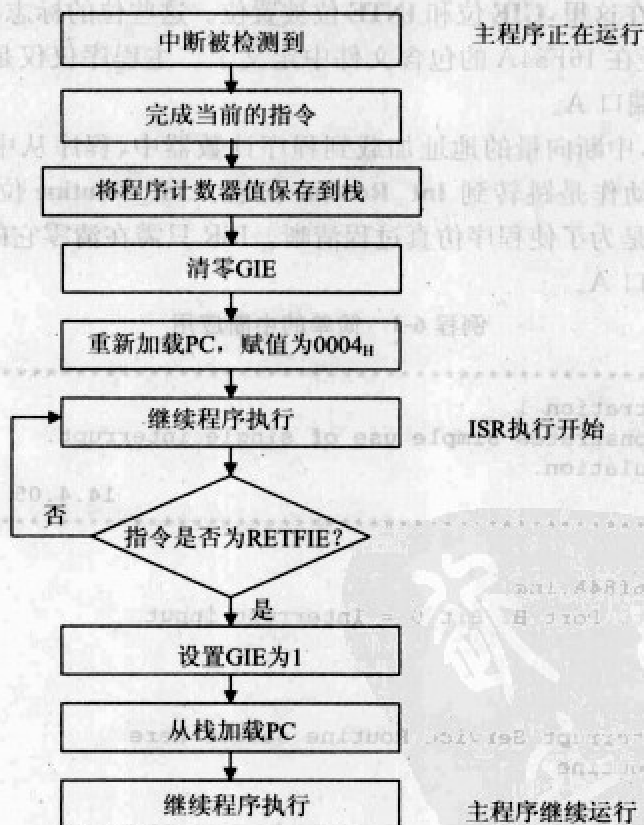


图 6-4 16F84A 中,事件的中断响应过程

6.2 编写含有中断的程序

6.2.1 编写仅含一个中断的程序

编写一个仅含一个中断的简单程序相对比较容易如果想编写成功,必须要注意以

下几点：

- ☐ ISR 必须从地址为 0004 的中断向量处开始；
- ☐ 通过设置 **INTCON** 寄存器中的启用位，启用将要用到的中断；
- ☐ 设置全局启用位 **GIE**；
- ☐ 在 ISR 中清零中断标志；
- ☐ 使用 **retfie** 指令结束 ISR；
- ☐ 确保已经配置好中断源（例如端口 B 或 Timer 0）以产生中断。

例程 6-1 是一个非常简单的中断例子，可在练习 6-1 中用于仿真。在该程序中，遵守了上述注意事项。程序像往常一样从复位向量 0000 处开始执行。但是，现在也使用了中断向量。程序的第一个动作就是从复位向量分支到 **start** 位置处，在 **start** 处开始程序的初始化。在这里，**GIE** 位和 **INTE** 位被置位。这些位的标志 **GIE** 和 **INTE** 可以使用，因为它们已经在 16F84A 的包含文件中定义了。主程序仅仅是循环输出位组合格式 $0A_H$ 和 15_H 到端口 A。

当中断发生时，中断向量的地址加载到程序计数器中，程序从中断向量处开始执行。ISR 的第一个动作是跳转到 **Int_Routine** 位置。**Int_Routine** 位于程序存储器的 0080_H 存储单元，这是为了使程序仿真过程清晰。ISR 只需在清零它的中断标志和返回主程序之前清零端口 A。

125

例程 6-1　简单的中断应用

```
;*************************************************************
;Interrupt Demonstration 1
;This program demonstrates simple use of single interrupt.
;Intended for simulation.
;Int_Demo1                                            14.4.05
;*************************************************************
;
      #include p16f84A.inc
;Port A all output. Port B: Bit 0 = Interrupt input
;
      org 00
      goto start
       org 04 ;Interrupt Service Routine starts here
       goto Int_Routine
      org 0010
;Initialise
start bsf     status,rp0   ;select bank 1
       movlw 01
      movwf trisb            ;portb bits 1-7 output
                                    ;bit 0 input
       movlw 00
      movwf trisa            ;porta bits all output
      bcf    status,rp0   ;select bank 0
       bsf    intcon,inte   ;enable external interrupt
       bsf    intcon,gie    ;enable global int
```

```
;Remove semi-colon from following instruction to change
                                        ;interrupt edge
;       bsf    option_reg,intedg
wait    movlw 0a    ;set up initial port output values
        movwf porta
        movlw 15
        movwf porta
        goto  wait
;
        org 0080
Int_Routine         ;Interrupt Service Routine continues here
        movlw 00
        movwf porta
        bcf   intcon,intf
        retfie
        end
```

仿真练习 6-1

从本书附属资源中复制程序 Int_Demo1 到 MPLAB®，然后给它创建一个项目。构建项目并启用仿真器。打开 Watch 窗口，在窗口中加入 **PORTA**、**PORTB**、**INTCON** 和 **PCL**。打开 Hardware Stack 窗口(在 **View** 下拉菜单下)，以观测栈的值。打开 Stimulus Controller，将 Pin RB0 设置为 Toggle。单步执行程序，并观察和理解每个所观察变量的值在每条指令下的变化。现在选中 RB0 引脚，单击 Fire 按钮，设置 RB0 为高，然后继续单步执行程序。这不会对程序执行产生任何改变，因为中断沿响应是下降沿触发的(**INTEDG** 位已被设置为一个复位值 0)。再次选中 RB0 引脚，单击 Fire 按钮，当你继续单步执行时，应当会触发一个中断序列。观察 Hardware Stack 的变化，程序执行跳转到 ISR，在执行完 ISR 后，程序从它被中断处的后面一条指令开始重新执行。试着通过改变 **INTEDG** 位，如程序中所示，来改变中断响应的时钟沿。

当你已经成功地实现了上述程序时，试着在程序中加入下述错误，通常初学者会犯这些错误，观察这些错误对程序造成的影响。

- ☐ 去掉 **bcf intcon,intf** 指令，未能对中断标志清零。
- ☐ 使用 **return** 而不是 **retfie**，未能正确终止 ISR。

6.2.2 编写含有多个中断的程序——识别中断源

通常，在程序中使用一个中断是非常简单的，但需要提醒大家的是，一旦我们开始在程序中使用多个中断，各中断之间会以非常复杂的方式相互影响。其复杂性几乎是程序中所使用中断数量的平方。

在前面我们学到，16F84A 有 4 个中断源，但只有一个中断向量。因此，如果多个中断启用，在 ISR 开始处到底哪个中断会发生。在这种情况下，程序员必须编写 ISR，在 ISR 开始处检测所有可能中断的标志，并由此确定该响应哪一个中断。例程 6-2 是采用这种方式的一段程序代码例子，在该段代码中假定 4 个中断源都启用。

126

例程 6-2 中断源识别

```
interrupt btfsc intcon,0         ;test RBIF
        goto portb_int
        btfsc intcon,1           ;test external interrupt flag
        goto ext_int
        btfsc intcon,2           ;test timer overflow flag
        goto timer_int
        btfsc eecon1,4           ;test EEPROM write complete flag
        goto eeprom_int

portb_int
...
place portb change ISR here
...
              bcf    intcon,0   ;and clear the interrupt flag
              retfie

ext_int
...
place external interrupt ISR here
...
              bcf    intcon,1   ;and clear the interrupt flag
              retfie
timer_int
...
place timer overflow ISR goes here
...
              bcf    intcon,2   ;and clear the interrupt flag
              retfie
eeprom_int
...
place EEPROM write complete ISR here
...
              bcf    eecon1,4   ;and clear the interrupt flag
              retfie
```

6.2.3 阻止中断对程序的破坏——保存上下文

由于中断可以在任何时候发生，因此它能够对程序执行产生极大的破坏，例程 6-3 证明了这一点。在例程 6-3 中使用了一个 16 位加法子例程。按照程序例程本来的目的，16 位数 9999_H 加上它自身，期望得到 17 位结果 13332_H。在子例程中，2 个低字节 **qlo** 和 **plo** 相加。产生的进位加到其中一个高字节上。然后，2 个高字节相加。在例程中编写了一个 ISR，像大多数 ISR 一样，该 ISR 能够改变进位标志和 W 寄存器的值。

127

如果中断在子例程的第一条 **movf** 指令之后立即发生，这条指令将 **plo** 的值保存在 W 寄存器中。由于 ISR 改变了 W 寄存器的值，当程序返回到子例程重新执行时，使用的是错误的 W 寄存器值。如果中断在第一条 **addwf** 指令后立即发生。进位位的值对于加法的正确执行很重要，但是此时在 ISR 中却丢失了进位位的值。

128

例程 6-3 中断的影响

```
;******************************************************************
;Int_context
;This program demonstrates the need for context saving.
;Intended for simulation.

;TJW 15.4.05                                    Tested 17.4.05
;******************************************************************
;Port A not used. Port B bit 0 used for ext.interrupt ip.
        #include p16f84A.inc
;
rhi             equ     10
rlo             equ     11
phi             equ     12
plo             equ     13
qhi             equ     14
qlo             equ     15

                org 00
                goto start
                org 04 ;here if interrupt occurs
                goto Int_Routine
;
start           org 0010
                bsf     intcon,inte ;enable external interrupt
                bsf     intcon,gie  ;enable global int
loop            movlw 99
                movwf phi     ;preload numbers to be added
                movwf plo
                movwf qhi
                movwf qlo
                call Double_add
                movlw 00      ;clear result
                movwf rhi
                movwf rlo
                goto  loop
;This subroutine adds two 16-bit numbers, stored in phi-plo, and qhi-qlo,
;and stores result in rhi-rlo. 16-bit overflow in Carry flag at end.
Double_add
                movf  plo,0          ;move plo to the W reg
                addwf qlo,0          ;add lower bytes
                movwf rlo
                btfsc status,0
                incf  phi,1          ;add in Carry
                movf  phi,0
                addwf qhi,0          ;add upper bytes
                movwf rhi
                return
Int_Routine
                bcf   status,0       ;clear the Carry flag
                movlw 0ff            ;change W reg value
                bcf   intcon,intf
                retfie
                end
```

仿真练习 6-2

从本书附属资源中复制程序 Int_Context 到 MPLAB®，然后给它创建一个项目。构建项目并启用仿真器。打开 Watch 窗口，在窗口中加入 **qhi**、**qlo**、**phi**、**plo**、**rhi**、**rlo**、**STATUS** 和 **WREG**。打开 Stimulus Controller，设置 Pin RB0 为 Toggle。单步执行程序，并检测加法是否正常工作以及是否得到期望的结果。现在试着在程序中的不同位置引入中断。注意中断在这些位置发生后，ISR 执行是如何破坏加法结果的正确性的。

在 CPU 中的一个特殊的程序行为中使用的临时数据叫做它的上下文(context)。在 PIC 16 系列中，上下文至少包括 W 寄存器和状态寄存器。很明显，在中断发生时保存上下文非常重要。一些微控制器自动保存上下文，但是 PIC 16 系列微控制器不能自动保存。因此，程序员必须确保在程序中需要保存什么样的上下文。

例程 6-4 显示的是 Microchip 公司推荐的保存上下文的方法，它保存 W 寄存器的值到预先指定好的存储单元 **W_TEMP** 中并保存状态寄存器的值到存储单元 **STATUS_TEMP** 中。使用的指令是 **swapf** 和 **movwf**，因为它们不会影响任何状态寄存器位。

129

例程 6-4 保存上下文

```
PUSH    movwf  w_temp           ;Copy W to W_TEMP register,
        swapf  status,0         ;Swap status to be saved into W
        movwf  status_temp      ;Save status to STATUS_TEMP register
ISR                             ;Interrupt Service Routine
...
        actual ISR goes here
...
POP     swapf  status_temp,0    ;Swap nibbles in STATUS_TEMP register
                                            ;and place result into W
        movwf  status           ;Move W into STATUS register ;sets bank to original
                                            ;state
        swapf  w_temp,1         ;Swap nibbles in W_TEMP and keep result in W_TEMP
        swapf  w_temp,0         ;Swap nibbles in W_TEMP and place result into W
...
        clear interrupt flag(s) here
...
        retfie
```

仿真练习 6-3

改写例程 6-4 中保存上下文的方式，并将它加入到 Int_Context(例程 6-3)中。需要为 **w_temp** 和 **status_temp** 定义存储单元。检测是否无论在哪里强制发生中断，程序都能正常运行。

6.2.4 阻止中断对程序的破坏——临界区域和中断屏蔽

对于发生在与前面所讨论的子例程类似的程序段中的中断问题，我们可以通过适当地上下文保存来解决其中的一部分。但遗憾的是，我们不能解决所有的问题。要解决各种情况的中断发生的问题，不能仅采用保存上下文这一种方法。

如果中断在软件延时程序中发生，例如例程 5-2 的软件延时程序，那将会怎么样？由于 ISR 的执行需要一定时间，延时的长短将会增加，这会对程序行为造成很大的破坏，再怎么保存上下文也不能解决这一问题。

考虑一个更细节的问题。例程 6-5 中的 ISR 获取保存在 **rhi-rlo** 中的字（在例程 6-3 的子例程中计算得到），并将它输出给连接到端口 A 和 B 的 12 位的数模转换器（DAC）。假设整个程序限制 **rhi-rlo** 中的字为 12 位，且在例程 6-3 的子例程执行过程中，中断发生，例程 6-5 中的 ISR 开始执行。在程序中实现了上下文的保存，那么还会有什么问题吗？

然而，这里有一个问题。ISR 正在使用它中断的程序所计算的结果。假设在中断发生时，**rlo** 的值已经更新而 **rhi** 还没有。ISR 输出 **rlo** 的新值和 **rhi** 的旧值。两者合在一起构成的是一个毫无意义的数，这就造成一个潜在的错误结果。

130

例程 6-5　使用程序中计算的数据的中断

```
Int_Routine
      movwf W_temp         ;Copy W to TEMP register,
      swapf status,0       ;Swap status to be saved into W
      movwf status_temp    ;Save status to STATUS_TEMP register
      bcf   status,5       ;ensure we are in Bank 0
movf  rhi,0           ;output higher 4 bits to DAC
movwf porta
movf  rlo,0           ;output lower 4 bits to DAC
movwf portb
swapf status_temp,0   ;Swap nibbles in STATUS_TEMP register
                      ;and place result into W
movwf status          ;Move W into STATUS register ;set bank to original
                      ;state
swapf W_temp,1        ;Swap nibbles in W_TEMP and place result in W_TEMP
swapf W_temp,0        ;Swap nibbles in W_TEMP and place result into W
bcf   intcon,intf
retfie
```

所以，我们必须接受这样一个事实，即：在某些程序区域内，无论是否保存上下文，在任何情况下都不希望有中断发生。我们把这些程序区域叫做临界区域（critical regions）。在临界区域执行时，可通过改变 **INTCON** 寄存器中的启用位来禁止或屏蔽中断。临界区域通常包括所有时间敏感的程序行为和对 ISR 所用的结果的计算。时间敏感的程序行为本身就包括延时循环和输出的多指令配置。

适当使用保存上下文和临界区域这 2 种技术，我们就可以很好地使用中断了，而且不会存在中断本身可能带来的对程序的破坏。

6.3 计数器和定时器概述

6.3.1 数字计数器回顾

使用触发器做一个数字计数器是非常简单的。计数器可以做成递增计数、递减计数,还可以清零和预置数并且可以通过计数器输出的溢出与其他计数器级联构成一个更大范围的计数器。图 6-5 是一个简单的计数器例子。8 个负时钟沿触发的 J-K 双稳态触发器连接在一起,每个触发器的输出 Q 都是下一个触发器的时钟输入。由于 J 和 K 的值都固定为逻辑 1,在每个输入负沿时触发器翻转。该计数器产生 8 位的二进制数,该 8 位二进制数由 8 个双稳态触发器的共 8 个输出 Q 组成,Q_7 是最高有效位,而 Q_0 是最低有效位。在每个输入时钟周期的负沿时,计数器递增加 1。

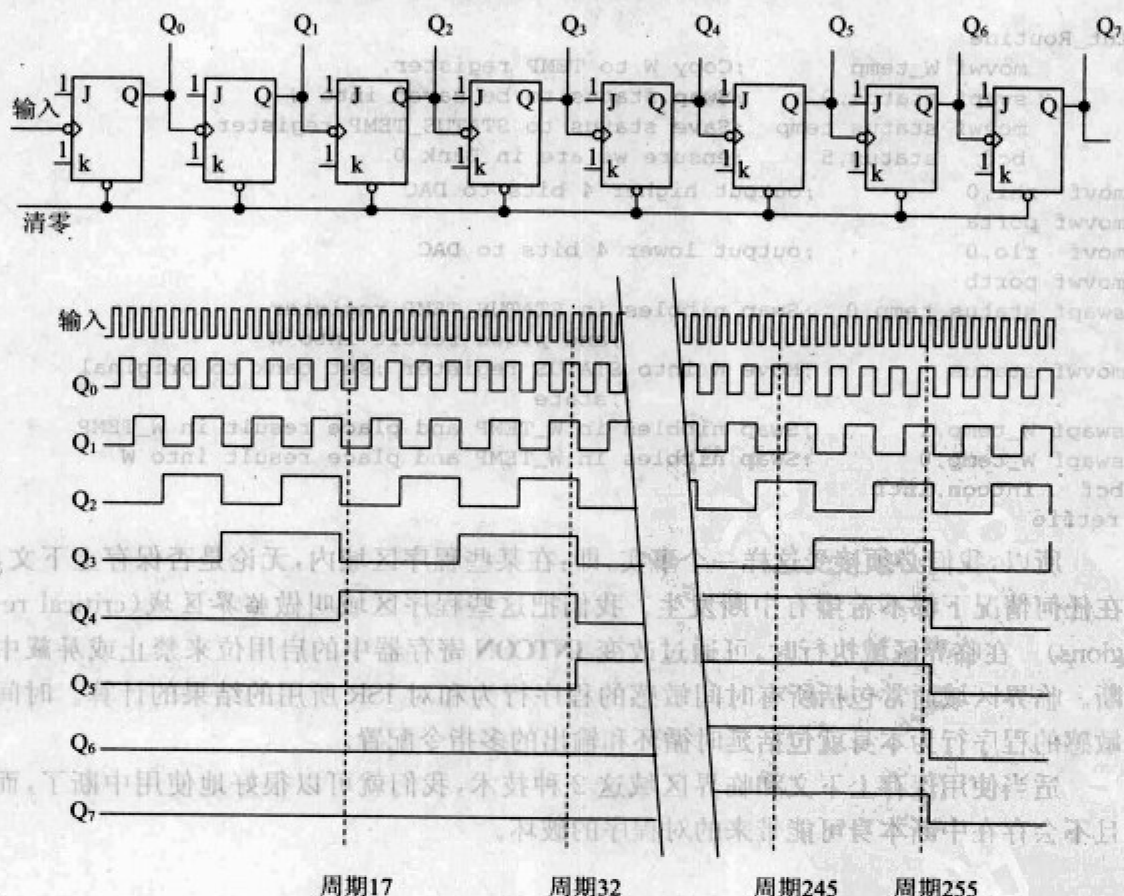


图 6-5 由 8 个触发器构成的数字计数器

图 6-5 中下半部分显示的是输出时序图。从图中可以看到,在一个输入周期后,Q_0 已经变为逻辑 1。在 16 个输入周期后(也就是在周期 17 中),计数器产生一个 8 位的

字 00010000_B，即 16_D，而在 31 个周期后，产生的字为 00011111_B，即 31_D。当 255 个输入周期完成后，计数器保存的字为 11111111_B，即 FF_H。如果再出现一个输入周期，那么所有的触发器都向前传送 0，并且输出回到 00000000_B。Q_7 的负沿可以用于标识计数器已经溢出。

131

如果清零线激活，图 6-5 的计数器可以复位为 0。如果给计数器加上预置功能，用于给计数器预先指定一个任意的数，会稍微复杂一点。这样做，我们可以得到一个功能多样的数字子系统，它是微控制器的计数器的基本构成，如图 6-6 所示。仅和计数器相连的重要的互连线是时钟输入、溢出输出和 8 位的读取或加载数据线，该数据线可以通过门控共享一个双向数据通道。

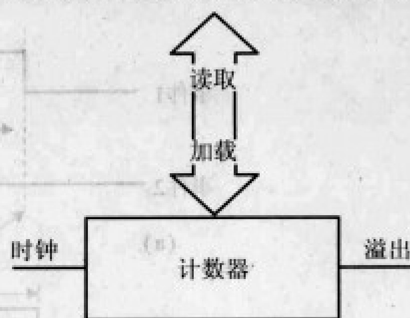


图 6-6　数字计数器的框图

6.3.2　将计数器用作定时器

微控制器具有计数的能力是相当有用的，例如：计算在传送带上传送的小物品个数、计算向自动贩卖机中投入硬币的个数，或计算通过一扇门的人数。然而，如果微控制器具有计算时间长短的能力，会更加有用。计数器能够做到这一点。

假设图 6-5 的输入信号是 1ms 稳定频率的时钟。那么计数器就会非常精确地每秒加 1。在 16 个时钟周期之后，恰好过了 16ms；在 31 个周期之后，恰好过了 31ms；等等。可选择在某个时刻开始输入时钟，所以就可以计算从该时刻到某个时间的长短。计算时间长短的分辨能力由时钟的周期决定。在本例中，分辨率是 1ms，我们不能计算小于它，或它的几分之一的时间的长短。此外，对于 1ms 的输入周期，8 位计数器在溢出前可以计算到 255ms。将计数器用作定时器非常重要，因此通常把计数器叫作计数器/定时器（counter/timer，C/T），或者为了突出它的重要性，就简单地叫作定时器（timer）。

132

计数器/定时器的一个明显的应用是计算 2 个“事件”之间的时间。这 2 个事件或许都是从外部产生。可能的一种情况是，微控制器产生第 1 个事件，而第 2 个事件作为响应在稍后一段时间发生。计算 2 个脉冲之间的时间或者单个脉冲的持续时间也是有必要的。图 6-7 显示了通常的计时情况。从图中看起来实际的计时很简单——在第 1 个事件发生时启动计数器/定时器，而在第 2 个事件发生时停止它。事实上，要做到这样的计时是非常具有挑战性的。要想精确地计算 2 个事件之间的时间，计数器/定时器的启动和停止就必须与事件的发生完全同步。要做到这一点，最好的方式是使用中断。如果没有中断，我们将不得不连续不断地扫描输入去检测事件在什么时候发生——在这种情况下，使用计数器/定时器就不值得了，因为我们也可以在软件中来实现定时。如果在两条不同的线上有 2 个外部事件发生，那么我们仍会遇到问题，因为在 PIC 16 系列中仅有一个外部中断。

133

对这种情况下的时间计算，我们将会在第 10 章看到一个非常好的例子。在第 10

章，我们还会学到计数器/定时器的一些增强功能，这些功能能解决计数器/定时器的启动和停止与要测量的事件之间的精确同步的问题。

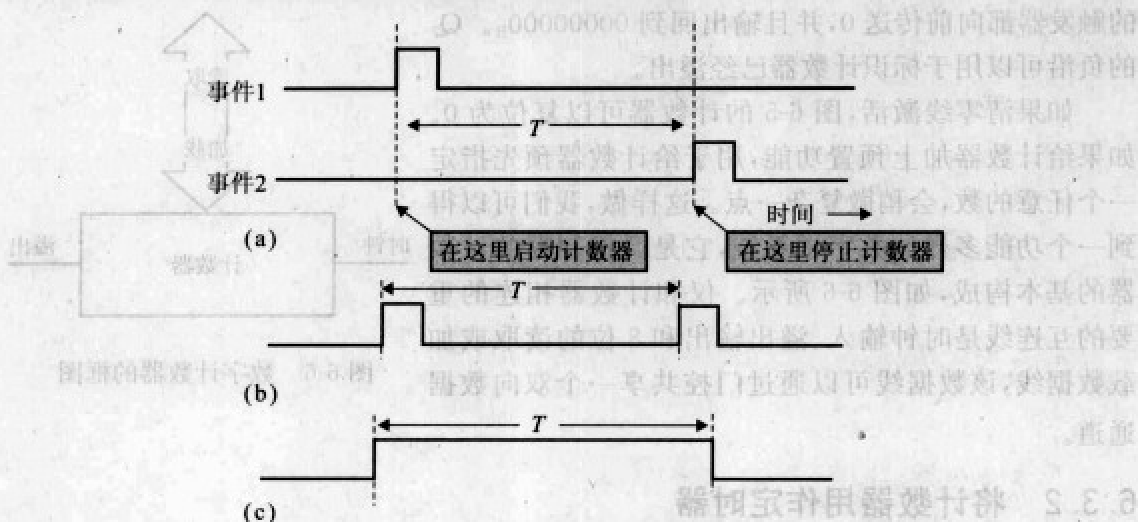


图 6-7 时间测量面临的挑战

6.3.3 16F84A Timer 0 模块

16F84A Timer 0 是较小微控制器中的一个典型的简单计数器/定时器。在 Timer 0 中使用了一个与图 6-5 相同的 8 位计数器，该计数器是存储器映射中的一个 SFR，与 Timer 0 的其他部分相连，同时在 Timer 0 中加入了一些额外的有用的功能，将这些功能和计数器封装在一起，就构成了 Timer 0。Timer 0 的框图见图 6-8。图 6-8 中有一个实际的 8 位计数器，标有 **TMR0**。从图 2-5 中可以看到，它是作为位于 Bank 0 中的存储单元 01 处的 **TMR0** 寄存器出现的。同所有性能优异的微控制器外围设备一样，Timer 0 可配置，可通过 **OPTION** 寄存器中的多个位来控制，如图 6-9 所示。

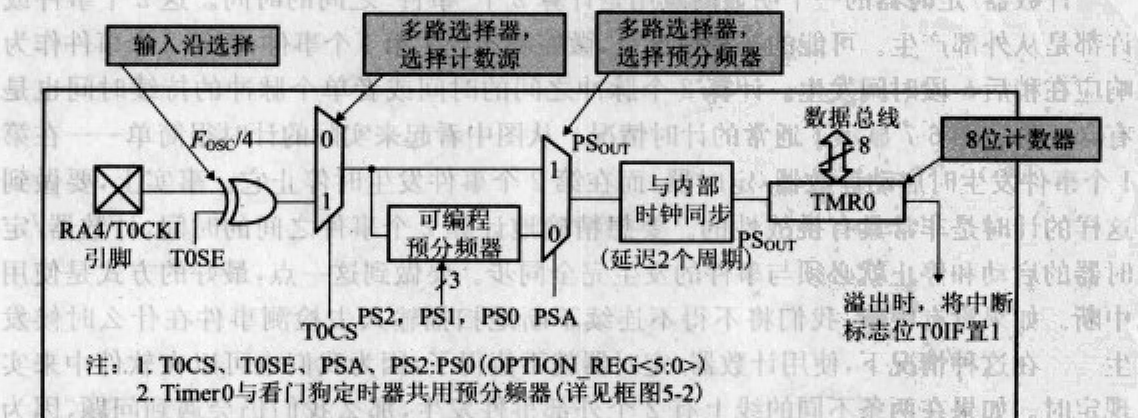


图 6-8 16F84A Timer 0 模块（阴影框中所附标签为作者所加）

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
RBPU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0
位 7							位 0

位 7　**RBPU**：PORTB上拉启用位

1=禁止PORTB上拉

0= 通过单独的端口锁存器值启用PORTB上拉

位 6　**INTEDG**：中断信号沿选择位

1=RB0/INT引脚上升沿中断

0=RB0/INT引脚下降沿中断

位 5　**T0CS**：TMR0时钟源选择位

1=RA4/T0CKI引脚电平变化

0=内部指令周期时钟(CLKOUT)

位 4　**T0SE**：TMR0时钟源沿选择位

1=RA4/T0CKI引脚电平由高到低转变时递增

0 =RA4/T0CKI引脚电平由低到高转变时递增

位 3　**PSA**：预分频器分配位

1=将预分频器分配给WDT

0= 将预分频器分配给Timer 0模块

位 2~0　**PS2:PS0**：预分频比选择位

选择位的值	TMR0分频比	WDT分频比
000	1:2	1:1
001	1:4	1:2
010	1:8	1:4
011	1:16	1:8
100	1:32	1:16
101	1:64	1:32
110	1:128	1:64
111	1:256	1:128

图 6-9　16F84A OPTION 寄存器

从图 6-8 的左边可以看到 **TMR 0** 计数器有两个可能的时钟输入源。其中一个是 RA4 引脚(即 16F84A 的引脚 3，见图 2-1)。另一个引脚是内部指令周期的频率，标有 *Fosc*/4。通过 **T0CS** 控制的多路选择器来选择一个时钟源，**T0CS** 位于 **OPTION** 寄存器中。外部输入通路包含一个异或门，用于反转输入信号，反转功能通过 **T0SE** 位控制。第 1 个多路选择器的输出在到达第 2 个多路选择器之前就分支了。第 2 个多路选择器选择一条直接通路，或者一条经过可编程预分频器的通路。选择哪一条通路由 **OPTION** 寄存器中的 **PSA** 位控制。在这里有些复杂的是，实际电路中 Timer 0 与看门狗定时器(Watchdog Timer，WDT)共用预分频器，我们将在本章的后面学到 WDT。现在 [134]

我们只需要知道,如果 **PSA** 设置为 1,那么预分频器分配给 WDT,而多路选择器选择没有预分频器的输入通路。预分频器本身受 **OPTION** 寄存器的 **PS2**、**PS1** 和 **PS0** 位控制。观察图 6-9 中这几位的情况,就会发现,通过这几位可以选择对输入的时钟信号进行不同的分频。第 2 个多路选择器的输出与内部时钟同步,然后作为实际计数器的输入。当计数器溢出时,设置定时器溢出标志,它是 PIC 微控制器的 4 个中断源中的一个,如 135 图 6-2 所示。

6.4　16F84A Timer 0 的使用——以电子乒乓球游戏为例

像 Timer 0 这样简单的计数器/定时器可用在许多应用中。接下来我们看 2 个例子,这 2 个例子都是基于电子乒乓球游戏程序的,并易于仿真。

6.4.1　对目标或事件计数

Timer 0 的最简单应用是用作计数器,计算通过外部输入进入微控制器的脉冲个数。从电子乒乓球游戏的电路(附录 2,图 A2-1)中可以看到,右边"球拍"按钮连接到 16F84A 的引脚 3 上。例程 6-6 所示的程序是一个非常简单的计数例子,它在适当的时候启用计数器,并使用右边按钮作为计数器的输入,连续不断地在与端口 B 相连的 LED 上显示当前值。

我们选择外部输入(即 **T0CS**=1)来配置 Timer 0。选择在输入信号的哪个沿触发并不太重要,但是考虑到存在开关反弹的风险,我们选择与开关释放相关的输入信号沿,也就是上升沿来触发 Timer 0,因为此时它几乎没有反弹的可能。所以,**T0SE**=0。我们不需要预分频器,因为我们希望统计按钮开关按下的实际次数,所以,**PSA**=1。因此,也不必关心 **PS2**、**PS1** 和 **PS0** 的值(因为在该应用中没有用到 WDT)。在本段中所有未被提到的 **OPTION** 寄存器位对于电子乒乓球游戏程序来说是不重要的,都可以随 136 意设置为 0。最后,**OPTION** 寄存器配置的值为 00101000_B。

例程 6-6　对计算乒乓球连续对打次数的 Timer 0 的初始化

```
;************************************************************************
;cntr_demo                                          Counter Demonstration
;This program demos Timer 0 as counter, using ping-pong hardware
;TJW 15.4.05                                               Tested 15.4.05
;************************************************************************
;Clock freq 800kHz approx (RC osc.)
;Port A 4    right paddle (ip) Counter input.
;       2    "out of play" led (op)
;Port B 7-0. "play" leds (all op)
;Interrupts not used
;Config Word: RC oscillator, WDT off, PU timer on, code protect off
;
```

```
        list p=16F84A
        #include p16f84A.inc
;
                org     00
; Initialise
                bsf     status,rp0     ;select memory bank 1
                movlw B'00011000'
                movwf trisa            ;port A according to above pattern
                movlw 00
                movwf trisb            ;all port B bits outout
                movlw B'00101000'      ;set up TMR0 for external input, +ve edge,
                                                                ;no prescale
                movwf TMR0             ;as we are in Bank 1, this addresses OPTION
                bcf     status,rp0     ;select bank 0
;
                movlw 04        ;switch on "out of play" led to show power is on
                movwf porta
loop            movf    TMR0,0  ;Continuously display Timer 0 on Port B
                movwf portb
                goto    loop
                end
```

该程序可以运行在电子乒乓球游戏硬件上，在这种情况下，每次按下右边的“球拍”按钮都会使在LED上显示的二进制加1。

仿真练习 6-4

从本书附属资源中复制程序 Cntr_Demo 到 MPLAB®，然后给它创建一个项目。构建项目并启用仿真器。打开 Watch 窗口，在窗口中加入 **PORTB** 和 **TMR0**。打开 Stimulus Controller，设置 Pin RA4 为 Pulse high，脉冲宽度为1个周期。连续单步运行程序，单击 Fire 按钮，发出输入脉冲，并观察 Timer 0 和端口 B SFR 是如何递增计数的。

6.4.2 硬件产生的延时

在先前的电子乒乓球游戏程序中，使用软件产生的延时来记录LED点亮的时间。这种方式只能运用于简单的程序中，因为在执行软件产生延时时，CPU在整个延时过程中都不能做任何有用的事情。现在，我们可以有计数器/定时器可供使用，可以使用它来产生延时；而且如有必要，CPU还可以做其他事情，不需空闲。这看起来很简单，但是这里有一个小问题：我们怎么知道延时何时完成？如果需要不断地检测定时器的值来判断延时是否完成，那么我们在改进软件延时上就几乎没有进步。我们可以通过定时器溢出时中断(interrupt on overflow)来确定延时是否完成。如果中断的相关配置设置好，可以在延时结束时产生中断，我们就有一个非常有用的方法来产生有效的延时了。

第一步，将电子乒乓球游戏程序中的5ms软件延时子例程替换为受 Timer 0 控制的延时。内部时钟约为800kHz，所以指令周期的频率($Fosc/4$)为200kHz，也就是周期为5μs。现在在这个时钟频率下，Timer 0 可以在 255×5μs(即1275μs)内计数到它的

最大值(255),并在下一个周期(即 1280μs 之后)溢出。但是,我们在这里可以使用预分频器。如果输入的信号被除 4(即 **PS2**、**PS1** 和 **PS0** 设置为 001),那么 Timer 0 在 256×4×5μs(即 5.120ms)之后溢出。这与我们期望得到的 5ms 非常接近了,但是不够精确。

虽然电子乒乓球游戏中不需要精确的时序,但是假设我们需要一个非常接近 5ms 的延时,该怎么做呢?我们将输入的时钟除以 8,而不是 4,这就得到一个 25kHz 的分频,也就是周期为 40μs。现在 125 个 Timer 0 输入周期会得到 40×125μs(即 5.00ms)的延时,完全达到我们的 5ms 目标。如果我们按照这种方式来设置预分频器并在每次延时开始时预置 Timer 的值为 256～125(即 131_D),那么就可以得到在溢出中断时终止的延时。

例程 6-7　在 delay5 子例程中使用 Timer 0

```
...
;Initialise
      org    0010
start bsf    status,5         ;select memory bank 1
      movlw B'00011000'
      movwf trisa             ;port A according to above pattern
      movlw 00
      movwf trisb             ;all port B bits op
      movlw B'00000010'       ;set up TMR0 for internal input, prescale by 8
      movwf TMR0              ;as we are in Bank 1, this addresses OPTION
      bcf    status,5         ;select bank 0
...
...
;introduces delay of 5ms approx
      delay5 movlw D'131'             ;preload counter, so that 125 cycles, each
                                      ;of 40us, occur before timer overflow
      movwf TMR0
del1  btfss intcon,2                  ;test for Timer Overflow flag
      goto del1                       ;loop if not set
      bcf intcon,2                    ;clear Timer Overflow flag
      return
```

例程 6-7 中的程序代码实现了这一方法。它包括初始化代码段和改正后的延时子例程。中断未被启用,并且子例程通过检测溢出中断标志来确定什么时候延时完成。对于程序员来说,带来的好处是可以通过配置 Timer 0,而不是通过调整软件程序来得到延时。"溢出时中断"并未被启用,因为在这里并不需要用到它。然而,在更高要求的程序中,该中断需要启用,并且延时期间通常保证 CPU 的其他活动。

仿真练习 6-5

按照例程 6-7 中的改动修改电子乒乓球游戏程序。单击 **Debugger** > **Settings**,设置时钟频率为 800kHz。使用跑表来观察修改后的延时子例程的时间。注意 **call**、**return** 和定时器加载指令是怎么加到延时程序中的?你是否可以改进它以进一步提高它的精确性?

138

6.5 看门狗定时器

我们需要注意16F84A的另一个定时器,虽然这个定时器通常不会用于简单的应用中。这个定时器就是看门狗定时器(Watchdog Timer,WDT)。任何一个基于计算机的系统都会面临一个大的危险,那就是在某种情形下软件出错,而且系统死机或没有任何响应。在台式机中,死机是很令人讨厌的,通常我们会通过重新启动来解决。在嵌入式系统中,死机会造成很大的破坏,因为没有用户会注意到系统出现了错误,并且根本就没有用户接口。WDT对该问题提供了一个相当严厉的"解决方案"。WDT是微控制器内部的一个计数器,用来不断地计数。它一旦溢出,就会强制微控制器复位(见图2-11)。由程序员来保证在程序中WDT被反复清零。可以通过 **clrwdt** 指令来清零WDT。只有当程序不能正常运行时,对WDT清零的指令就不再执行,WDT溢出就会发生。

通常对于嵌入式系统来说,WDT复位并不是一个好消息,因为此时所有的当前设置都会丢失,且程序重新开始。但是,与程序根本不能运行相比,这种情况还好一点。需注意的是,WDT通过状态寄存器中的$\overline{\textbf{TO}}$位给它后面的程序行为留下了一个线索。可以在程序的起始处检测该位,以此来区别上电复位和WDT复位。

16F84A WDT通过配置字中其中一位来启用,见图2-6。这样,它可以计算或者不计算微控制器启动后运行的时间。WDT通过内部RC振荡器来驱动,WDT提供一个额定的18ms超时周期。但是在某种程度上,该值决定于温度和电源电压,并且不同器件的该值也不同。通过使用Timer 0的预分频器可以扩展超时周期,在这种情况下,超时周期可以扩展到128×18ms,也就是大约2.3s。

6.6 休眠模式

虽然在本章中我们考虑的是计时,但是在时间几乎暂停时,需要考虑微控制器操作的另一个重要方面——休眠模式。休眠模式是一种重要的减小能耗的方法。通过执行 **SLEEP** 指令可以将微控制器置于该模式,见附录1。一旦处于休眠模式,微控制器的动作行为暂停。时钟振荡器被关闭,WDT清零,程序执行暂停,所有端口保持它们的当前值,且状态寄存器(图2-3)中的$\overline{\textbf{PD}}$和$\overline{\textbf{TO}}$分别被清零和置位。一旦退出休眠模式,WDT可继续运行。在这种情况下,功耗降低到几乎可忽略的地步——参考文献2.1引证了在特殊理想的工作条件下,休眠模式的功耗典型值是1μA。

一旦进入休眠模式,需要一个显式的事件来唤醒微控制器。在下面几种情况下,16F84A会退出休眠模式。

- 通过MCLR引脚外部复位(External reset through MCLR pin)。当唤醒微控制器时,也会复位微控制器,所以看起来对它的使用受限于完成程序的重启。但是,可

以通过状态寄存器的$\overline{PD}$引脚的状态来检测微控制器是否正处于休眠模式。

- WDT 唤醒(WDT wake-up)。在休眠模式下,WDT 的功能会有点不同。从图 2-10中可以看到,在休眠模式下,WDT 不能产生复位。相反,在溢出时,它仅产生从休眠模式的唤醒信号,微控制器从休眠模式之后的指令开始继续执行程序。
- 中断的发生(Occurrence of interrupt)。如图 6-2 所示,不管全局中断启用处于什么状态,任何一个启用的中断都可以唤醒微控制器,使之退出休眠模式。但是,Timer 0 不能产生中断,因为在休眠模式下内部时钟是禁止的。

一旦唤醒,振荡器电路就会重启。对于任何晶体振荡器模式来说,这意味着 T_{OST} 定时器也被激活,如图 2-11 所示。在程序重新开始执行前,它必须完成它的计数。所以,像人一样,16F84A 也需要一定的时间来苏醒和完成准备工作。

对于在需要注意功耗的产品来说,休眠模式相当有用。绝大多数的器件在上电后并不需要持续不断地运行。在它们不使用时,将其置于休眠模式,可以动态地减少它们的功耗。

6.7 其他中断

中断几乎是每一个微控制器的基本特征,但是不同的微控制器中断的结构也大相径庭,其中一个不同之处是中断向量的使用方式。

基于 Atmel 8051 的微控制器有 6 个中断,如表 6-1 所示。与 PIC 16 系列共享单一的中断向量不同,它的每个中断都有自己的中断向量。所以,外部中断 0 的 ISR 的起始地址为 0003_H,而 Timer 0 溢出的 ISR 起始地址为 $000B_H$。由于在这里每个中断都有自己的中断向量,不需要像例程 6-2 所做的那样,在 ISR 的起始处选择中断标志来检测中断源。但是,即使有多个中断向量,如果 2 个中断同时发生,也有必要有一种在 2 个中断中选择一个的方法。所以,Atmel 器件给不同的中断区分优先次序,如表 6-1 所示。如果 2 个中断同时发生,高优先级的中断先被响应。同 16F84A 一样,当响应一个中断时,Atmel 仅保存程序计数器到栈中。

表 6-1 Atmel 中断源和中断向量地址

中断源	符号	优先级	中断向量地址
外部中断 0	IE0	1(最高)	0003_H
Timer 0 中断请求	TF0	2	$000B_H$
外部中断 1	IE1	3	0013_H
Timer 1 中断请求	TF1	4	$001B_H$
串行收发	RI+TI	5	0023_H
Timer 2 溢出,Timer 2 外部	TF2+EXF2	6(最低)	$002B_H$

对于中断向量,Freescale 68HC08 以及之前所有的摩托罗拉公司产品采用的是一种非常不同于寻常的方法。不同于前面中断向量是 ISR 的起始地址的方法,而是在存

储器中保存起始地址。Freescale 将这些中断向量放在存储空间最顶部，如表 6-2 所示。显然，在这里需要非易失性的存储器。每个中断向量为 16 位，所以它占据 2 个 8 位的存储单元。复位向量总是放在存储器映射的最顶部。在这个地址中存放程序实际的起始地址。在复位向量之后，按照优先级递减的次序放置中断向量。表 6-2 只列出了最开始的一小部分中断向量。对于用户来说，采用这种方法的好处是用户可以将 ISR 放在存储器映射中任何他们想要存放的位置，只要他们能在指定的中断向量中正确地放入他们的地址。68HC08 也与 PIC 16 系列和 Atmel 不同，在 68HC08 中，当中断发生时，将所有的 CPU 寄存器保存在栈中，包括累加器（相当于 PIC 的 W 寄存器）、索引寄存器的低字节、程序计数器（2 个字节）和条件码寄存器。

表 6-2 Freescale 68HC08 中断源和中断向量地址（不完全）

中断源	符号	优先级	中断向量地址
复位向量（低）	—	—	$FFFF_H$
复位向量（高）	—	—	$FFFE_H$
软件中断（低）	SWI	1	$FFFD_H$
软件中断（高）	SWI		$FFFC_H$
中断请求（低）	IRQ	2	$FFFB_H$
中断请求（高）	IRQ		$FFFA_H$

6.8 更多的了解——中断响应延时

使用中断是为了能够在中断发生时立即引起 CPU 的注意，但是实际上引起 CPU 的注意有多快呢？从中断发生到 CPU 响应它的时间叫做*中断响应延时*（interrupt latency）。该延时取决于硬件的某些情况，最终也取决于程序运行的特征。图 6-10 的时序显示了中端 PIC 型号是如何响应一个外部启用的中断的。中断本身可以看作 **INT** 引脚上的正向脉冲。它使得中断标志 **INTF** 被置位。在内部振荡器时钟的 Q1 周期，该标志被采样。一旦完成采样，CPU 就检测到该中断，之后的中断响应过程就如图 6-4 所示。需要 2 个空周期来将程序计数器保存在栈中，将 0004_H 加载到程序计数器中，以及从该地址取值。

仿真练习 6-6

再次打开 int_demo1 项目，单击 Debugger>Settings，设置时钟频率为 4MHz。启用 Stopwatch（在 Debugger 下拉菜单下），然后单步运行程序。观察 Stopwatch 是如何按照我们预测的方式更新所花的时间的。现在产生一个中断。注意 Stopwatch 是如何记录 2 个指令周期的中断响应延时的。在 ISR 的末尾，可以看到 **retfie** 指令也需要 2 个指令周期的延时。

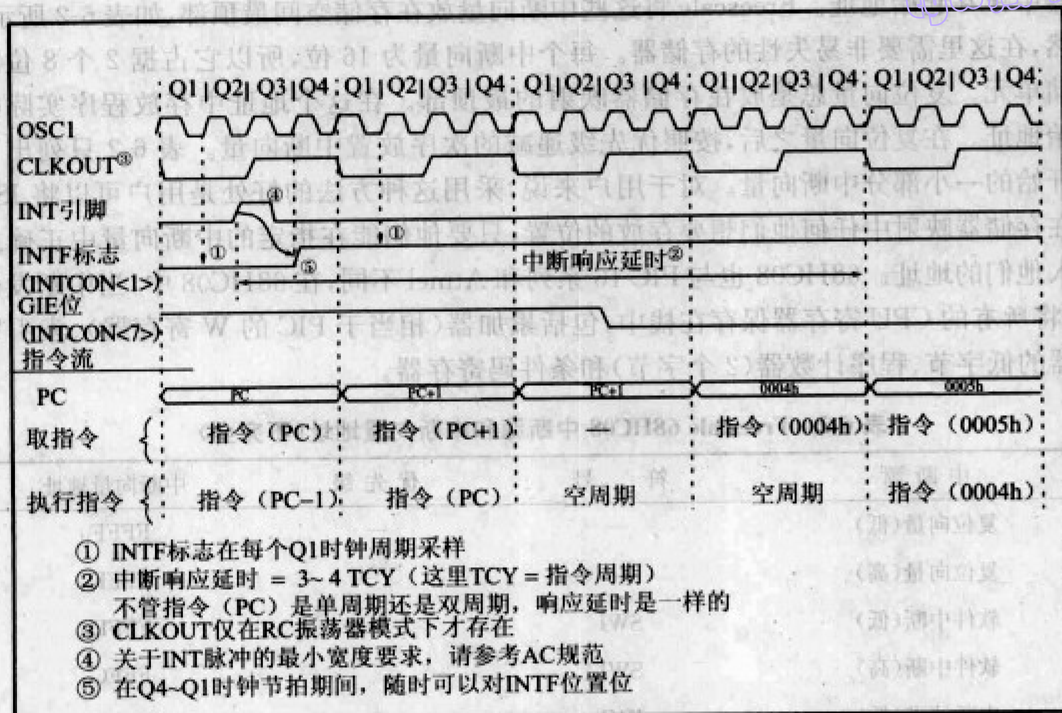


图 6-10 16F84A 的外部中断响应延时

小结

- ☐ 中断和计数器/定时器是几乎所有微控制器的重要硬件特征。
- ☐ 它们都含有大量重要的硬件和软件概念，这些概念都是必须要掌握的。
- ☐ 本章介绍了使用中断和计数器/定时器的基本技术。在更高级的应用中，它们的使用方式会更加复杂。

第三部分 较大的系统和 PIC® 16F873A

本部分由 5 章组成。通过学习，将逐步对微控制器的外围设备以及它们的基本原理有一个透彻的理解。本部分使用的微控制器核与前几章相同，但是采用了“大型”的 PIC 16 系列微控制器。本部分所介绍的外围设备也都可直接应用于 PIC 18 系列的微控制器。学习的重点在于了解这些外围设备以及如何在日益复杂的应用中灵活使用它们。本部分的例程使用汇编语言编写。

第 7 章
较大的系统和 PIC®16F873A

前面 5 章中，我们使用 PIC®16F84A 作为实例介绍了微控制器。这种型号的控制器以及其他类似的控制器对于较小的系统来说是个不错的选择。然而，有许多功能它们无法实现。因此，对于更复杂的应用，我们必须选择功能更强大的微控制器。“更强大”实际上意味着什么呢？回忆第 1 章中我们所述的微控制器的主要组成部分：

微处理器核＋存储器＋外围设备

改进其中任何一个因素都能使微控制器变得更“强大”。内含 CPU 的微处理器核可通过提高频率、改进内部结构或者指令集，使它的功能变得更强大。存储器可通过改进工艺、增大存储容量或者提高速度，使它的功能变得“更强大”。另外，也可以在微控制器中增加更多的外围设备或者对已有外围设备进行改进。

在嵌入式系统的许多应用中，最显著的性能提高不是由改进处理器核来获得的，而是通过增加新的外围设备（可能伴随着存储器的升级），这会给系统带来主要的性能改进。利用一些合适的外围设备、实时模/数变换、串行通信、功能复杂的定时部件可以很容易地提高系统的性能。

本书的第 7 章～第 11 章将继续学习 PIC 16 系统微控制器。但是，我们学习的是该系列中更复杂的 PIC 16F873A 微控制器。16F84A 与 16F873A 是同一系列的微控制器，因此它们的处理器核和指令集是相同的。但是，由于增加了许多性能很高的“新”外围设备，16F873A 的嵌入式控制功能获得了很大的提高。

这几章将通过 Derbot AGV 的应用来阐述一些概念。如果你没有构建 Derbot AGV 项目，也不用担心，在后面介绍许多新概念时，它仍然是一个很好的例子。本部分的程序继续使用汇编语言编写，但是我们将会体会到使用汇编语言以及一些 PIC 16 系列微控制器硬件特征的限制。因此，在本书的最后一部分，我们将在更高级的微控制器上使用 C 语言来开发复杂的多任务程序。

由于我们进行的是较大系统的设计，开发使系统正常并且稳定工作的工具非常重要。因此，本章会介绍一些新的、重要的调试工具和技术，它们将会在后面各章中用到。

在本章中你将学到：

- ☐ 1687XA 系列微控制器的结构，16873A 属于该系列；
- ☐ 1687XA 的存储映射和中断结构；
- ☐ 一些高级的调试嵌入式系统的工具；
- ☐ Microchip 公司电路内调试器的使用。

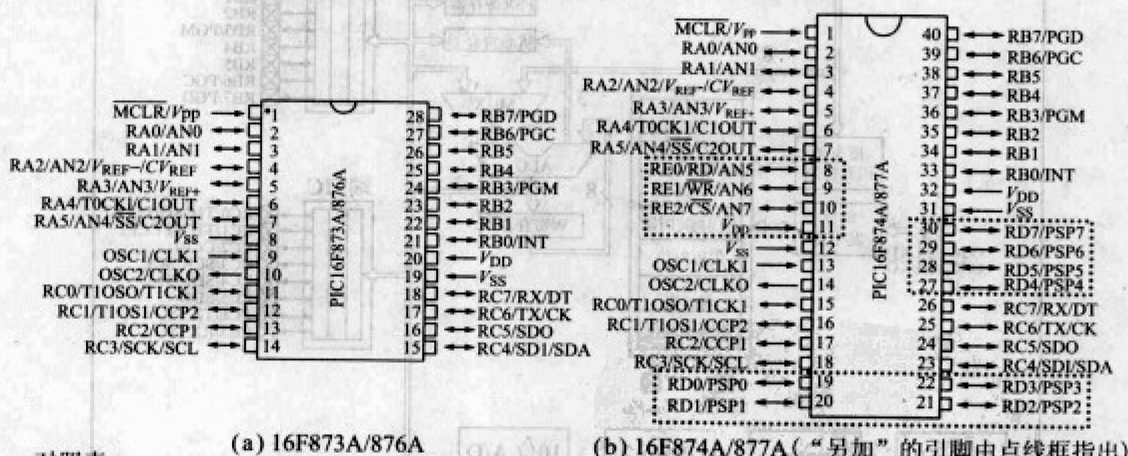
如果你准备构建 Derbot AGV 项目，本章将会给你提供一些构建该项目的第一步的指导，后面几章将继续给出一些指导。

7.1 PIC16F87XA 概述

第 2 章开头简要介绍了一个系列的微控制器，我们的实例微控制器 16F873A 是其中的一个分组。这个分组由 16F873A、16F874A、16F876A 和 16F877A 构成。一般称它们为 16F87XA。这个分组的每一个微控制器都有一个 LF 版本，例如 16LF873A 它可以在比标准设备使用的电压更低的电压下工作。

表 2-1 总结了该分组的特性。审查这个表，很容易发现这 4 个分组成员只是以不同的存储器大小和封装大小相互区别。较大的封装允许包含更多的并行输入/输出端口，引脚连接图 7-1 说明了这一点。封装较大的器件上“另加”的引脚由点线框指出。一个主要的区别是'874A/877A 新增加了端口 D 和 E。

本章后续部分一直到第 11 章都将详细介绍实例微控制器 16F873A。后面要构建的 Derbot AGV 项目中会使用到它，16F87XA 分组中的其他成员也会提到，特别是当它们具有'873A 不具备的特性时。



(a) 16F873A/876A

(b) 16F874A/877A（“另加”的引脚由点线框指出）

对照表：

RA0：端口A的引脚0。其他类似
RC0：端口C的引脚0。其他类似
RE0：端口E的引脚0。其他类似
AN0，AN1等：模拟输入通道
CCP：捕捉/比较/PWM
OSO/OSI：定时器1的振荡器输出/输入
SS：主从选择
V_{REF}：参考电压
DT/CK：同步串行数据/时钟(USART)
MCLR：主清零
PGC/PGD：电路内串行编程的时钟和数据
PGM：电路内串行编程的程序
RD/WR/CS：读，写，片选(并行从动端口)
RX/TX：异步串行接收/发送(USART)

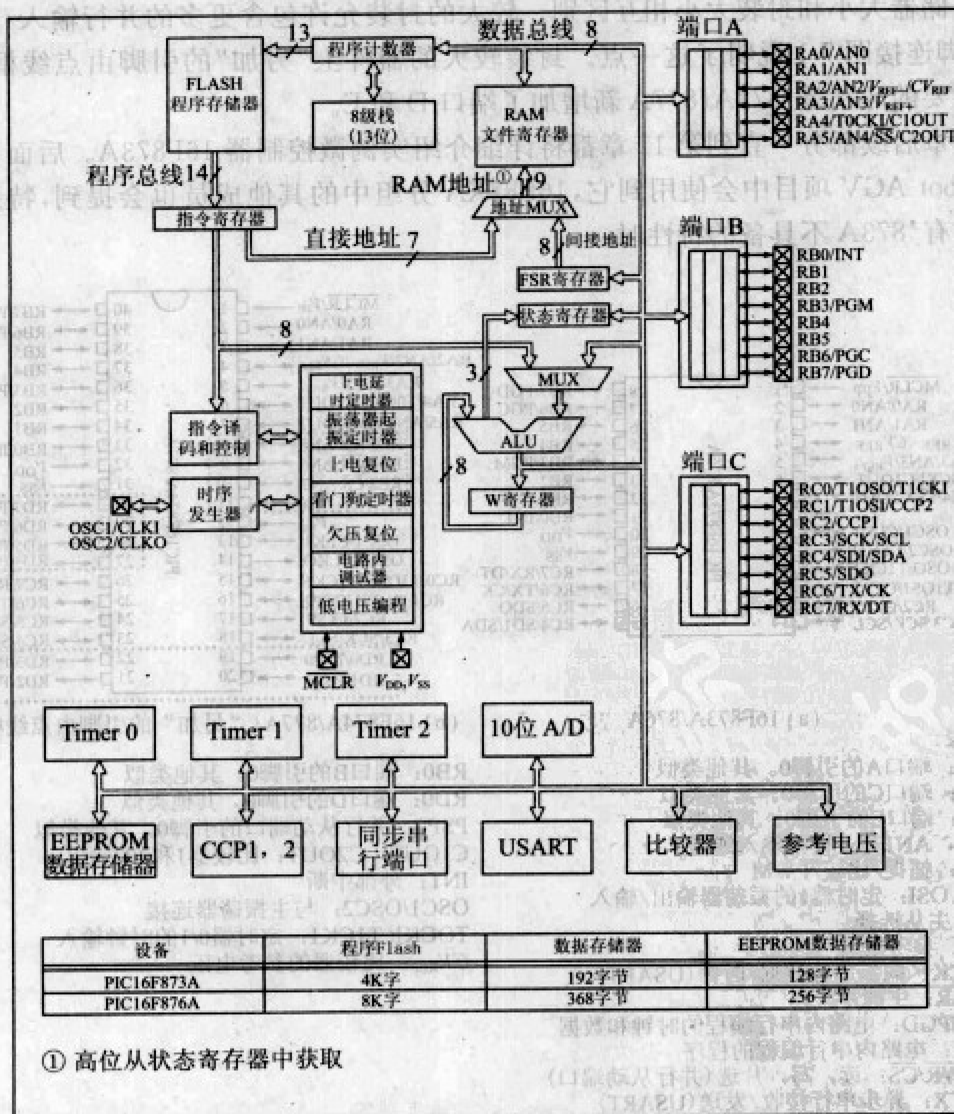
RB0：端口B的引脚0。其他类似
RD0：端口D的引脚0。其他类似
PSP0：并行从动端口的引脚0。其他类似
C1OUT，C2OUT：比较器1和2的输出
INT：外部中断
OSC1/OSC2：与主振荡器连接
TOCKI/T1CK1：定时器0/1的时钟输入
CV_{REF}：比较器的参考电压

SCK/SDI/SDO：串行时钟，串行数据输入，串行数据输出(同步串行端口，SPI模式时)
SCL/SDA：串行时钟，串行数据(同步串行端口，I²C模式时)

图 7-1 PIC16F87XA 的引脚连接图

7.2 16F873A 的结构图和 CPU

图 7-2 是 16F873A 和'876A 的结构图。这 2 款微控制器的区别在于存储器的容量,图底端的表列出了它们各自的容量大小。建议认真研究这个结构图,并将它们仔细地与图 2-2 所示的 16F84A 结构图进行比较。我们将会发现其中有些结构是相同的,有些结构只有轻微的改进,有些结构只是改变了图示方式,但是其中大部分的结构是新增加的。



设备	程序Flash	数据存储器	EEPROM数据存储器
PIC16F873A	4K字	192字节	128字节
PIC16F876A	8K字	368字节	256字节

① 高位从状态寄存器中获取

对照表:外围设备

A/D:模数转换器(analog-to-digital converter, ADC)　　CCP:捕捉/比较/PWM模块

USART:通用同步/异步接收/发送器

图 7-2 16F873A/876A 框图

7.2.1 CPU 和核

我们首先注意到 CPU 的结构同'F84A 是一样的，主要由 ALU、工作寄存器(Working register)和状态寄存器(Status register)组成。图中新增加了 3 条从 ALU 到状态寄存器的线，它们分别控制状态寄存器的 3 个状态位：Z、DC 和 C。在图 2-2 中省略了这 3 条线。

CPU 中的一个不同之处在状态寄存器上。在 16F84A 的状态寄存器中(图 2-3)，高两位未使用，位 5 用来对数据存储器的 2 个存储区进行选择。图 7-3 是 16F87XA 的状态寄存器。由于具有容量更大的数据存储器，状态寄存器的高三位都用来对数据存储器区进行选择。除此之外，状态寄存器保持不变。 146

R/W-0	R/W-0	R/W-0	R-1	R-1	R/W-x	R/W-x	R/W-x
IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C
位 7							位 0

位 7　**IRP**：寄存器组选择位(用于间接寻址)
1=区 2，3(100h~1FFh)
0=区 0，1(00h~FFh)

位 6和位5　**RP1**：**RP0**：寄存器组选择位(用于直接寻址)
11 = 区 3(180h~1FFh)
10 = 区 2(100h~17Fh)
01 = 区 1(80h~FFh)
00 = 区 0(00h~7Fh)
每组大小为128字节

位 4　$\overline{TO}$：超时溢出位
1 = 上电(power-up)后，执行CLRWDT指令，或SLEEP指令
0 = 发生WDT超时溢出

位 3　$\overline{PD}$：掉电位
1 = 上电后，或执行CLRWDT指令
0 = 执行SLEEP指令

位 2　**Z**：零位
1 = 算术或逻辑操作结果为0
0 = 算术或逻辑操作结果非0

位 1　**DC**：数字进/$\overline{借}$位(ADDWF、ADDLW、SUBLW、SUBWF指令)(对于借位，极性相反)
1 = 有第3位向第4位进位或无第3位向第4位借位
0 = 无第3位向第4位进位或有第3位向第4位借位

位 0　**C**：进/$\overline{借}$位(ADDWF、ADDLW、SUBLW、SUBWF指令)
1 =产生的结果的最高有效位(Most Significant Bit MSB)向前有进位
0 =产生的结果的最高有效位向前无进位
注：对于借位，极性相反。减法通过加上第2个操作数的二进制补码来执行
对于移位指令(RRF, RLF)，源寄存器的最高位或最低位移入此位

图 7-3　16F873A 状态寄存器

7.2.2 存储器

除了 CPU 结构与 16F84A 类似之外，16F873A/876A 的整个存储结构也基本保持不变，只是做了一些较小的调整。程序计数器(PC)是 13 位，因此可寻址 2^{13}(即 8192) 147~148

个存储单元。在 16F876A 中,这 13 位都用来寻址 8K 的程序存储器。但是,873A 只能寻址 4KB 的程序存储器。

图 7-3 中显示“直接地址”总线的宽度扩展到了 7 位。这个调整是为了适应容量变大的 RAM,这在下一节中讲述。然而,这个很明显的改变并不是对 PIC16 系列结构的“扩展”。这 7 位是从指令字(如图 4-13 所示)中得来的。从图 4-13 中可以看到指令字中有 7 位被保留用作“文件寄存器地址”,结构复杂的微控制器只是利用了这 7 位。

149

7.2.3 外围设备

16F84A 和 16F873A 都有 3 个相同(或者差不多相同)的外围设备:定时器 0、端口 A 和端口 B。这 2 款微控制器最大的不同在于新增的外围设备,它们可以从表 2-1 和图 7-2 中看出。本章将讲述并行端口,其他外围设备将在后续部分提到。

外围设备数目的增加同时也带来了 2 个挑战:这些外围设备如何与 CPU 交互,如何与片外器件交互?对于第 2 个问题,我们已经有了初步答案:16F873A 具有更多的引脚,而且许多引脚增加了共享的功能。为了与 CPU 交互,我们将会发现 16F873A 增加了许多特殊功能寄存器(Special Function Registers,SFR)和中断源。

16F873A 中增加了 3 个重要外围设备:欠压复位检测、电路内调试器和低电压编程,如图 7-2 所示。这些外围设备在本章后续部分会讲到。

7.3 16F873A 的存储器和存储器映射

16F873A 的存储结构与 16F84A 非常相似。但是它在存储容量上有很大的增加,这在前面已经提到过,它还有一些重要的技术进展,特别是电路内编程技术。

7.3.1 16F873A 的程序存储器

图 7-4 是 16F873A 的程序存储器的映射结构图。与图 2-4 相比较,可以发现它与 16F84A 唯一的不同是存储容量的大小。13 位的程序计数器可以寻址所有的存储器空间,但是很令人惊讶的是,图中有 2 个存储器页。对于 16F876A/877A 微控制器,它们甚至有 4 个存储器页。

程序存储器按这种方式分页的原因是程序的执行地址(即程序计数器)有多种产生方式。程序计数器的宽度为 13 位。低 8 位组成 **PCL** 寄存器,该寄存器属于特殊功能寄存器,可以像数据存储器单元一样被存取和处理。但是程序计数器的高 5 位不可读,可以通过写 **PCLATH** 寄存器的低 5 位来间接修改它。在写 **PCL** 的同时,**PCLATH** 的值被传递到程序计数器的高 5 位。

程序正常执行时,每条指令执行完毕后,程序计数器递增。但是,另外还有 3 种程序计数器的修改方式,它们会在程序中修改程序计数器的值。下面分别对它们进行介绍,如图 7-5 所示。

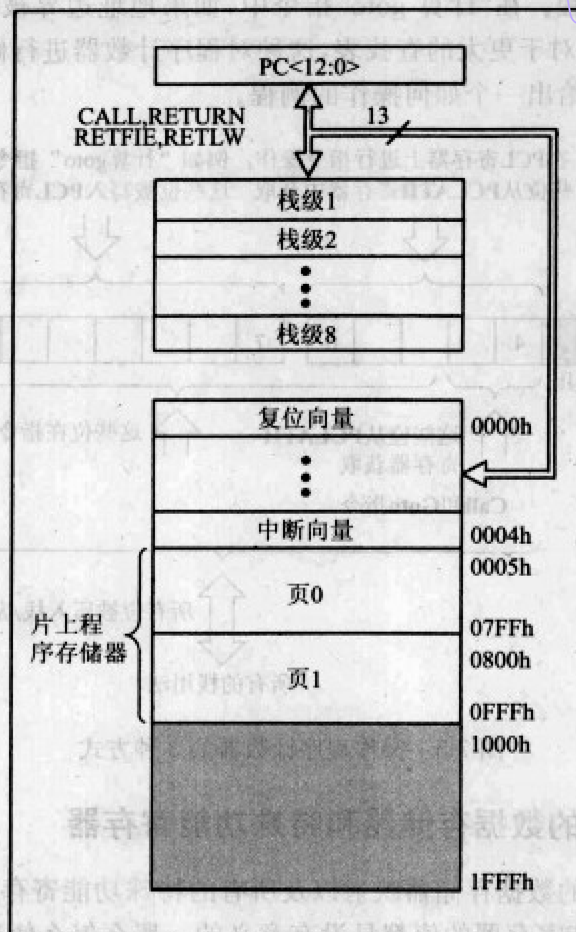


图 7-4 PIC 16F873A/874A 程序存储器映射和栈

1. 通过栈传输

栈宽度为 13 位，与程序计数器的位宽相同。因此，任何对栈进行操作的指令比如 **return** 指令将会触发 13 位的程序地址在栈和程序计数器之间传输——这样就没有必要考虑地址位的缺失了。

150

2. 通过 call 和 goto 指令

图 4-13 所示的指令字格式中，这 2 条指令格式只提供了 11 位的地址。这 11 位地址的寻址空间为 2^{11}(2K 字)，正好为图 7-4 程序存储器的一个页。当使用这 2 条指令时，另外 2 个“缺少”的地址位从 **PCLATH** 寄存器的位 4 和位 3 中获得。这就要由程序员来判断指令是否需要跨页，并正确设置 **PCLATH** 寄存器。数据手册中[7.1]包含了一个如何操作的例程。

3. 通过写 PCL 寄存器

在某些情况下，可以直接写 **PCL** 寄存器，如 5.4 节所述的“计算 goto”。现在在 **PCL** 寄存器中，只有程序计数器的低 8 位可以直接被修改，且程序员似乎在一个 256

151

字大小的页上进行编程。在“计算 goto”指令中，如果地址边界被跨越，就必须正确设置 PCLATH 寄存器。对于更大的查找表，这种对程序计数器进行修改的方式将是一个挑战。参考文献 5.1 给出一个如何操作的例程。

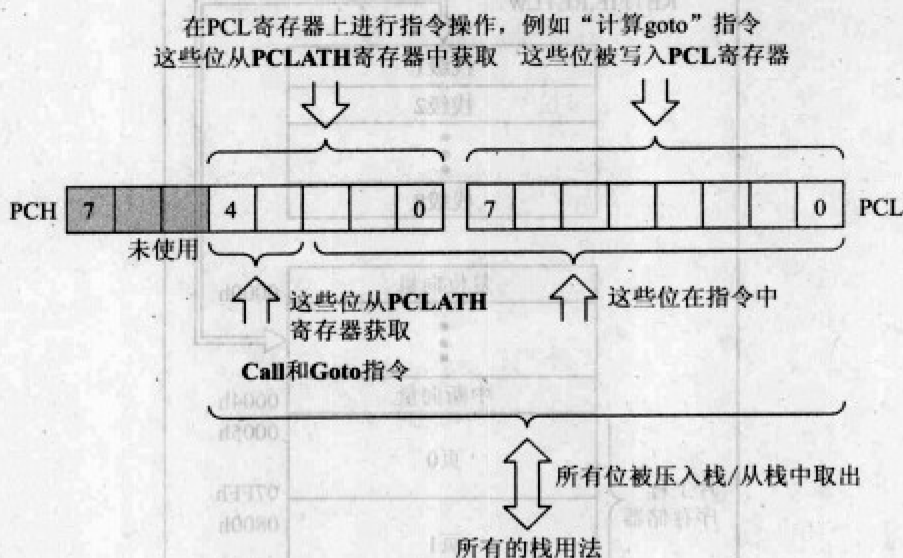


图 7-5 操作程序计数器的 3 种方式

7.3.2 16F873A 的数据存储器和特殊功能寄存器

16F873A/874A 的数据存储器映射以及所有的特殊功能寄存器如图 7-6 所示。最初，所有这些特殊功能寄存器的值都是没有意义的。那么怎么使这些寄存器代表特殊的含义呢？更令人头痛的是，大部分特殊功能寄存器都包含 8 个有效的比特位，每一位都具有一个特殊的功能，这些功能我们可能都需要去了解。不过不用担心，这里会使用详细的分步介绍的方法来让我们对这些庞杂的特殊功能寄存器有一个透彻的了解。本书的后续章节，将会介绍和使用其中大部分的特殊功能寄存器。

在结构上，存储器被划分成 4 个区。可以通过状态寄存器（如图 7-3 所示）的位 6 和位 5 来对这 4 个区进行选择。图 7-2 中 7 位的“直接地址”总线对应图 4-13 前面的 2 条指令格式。7 位的总线允许寻址 2^7(128) 个存储器单元，即一个区的存储容量。这 7 位宽的地址和状态寄存器中的 2 个区选择位连在一起构成图 7-2 中 9 位的 RAM 地址，它等价于图 7-6 中的“文件地址”。

前 2 个数据存储器区完全使用了区中所有的存储单元。例如，第一个区包含 32 个 SFR 和 96 个通用的存储器单元。在 16F873A/874A 中，对后 2 个区的使用是受限制的。这 2 个区没有通用存储器单元，并且大部分的特殊功能寄存器只是从其他区映射过来。核对一下哪些特殊功能寄存器是存储区 2 和 3 特有的。它们包括哪些？

0区	文件地址	1区	文件地址	2区	文件地址	3区	文件地址
间接地址*	00h	间接地址*	80h	间接地址*	100h	间接地址*	180h
TMR0	01h	OPTION_REG	81h	TMR0	101h	OPTION_REG	181h
PCL	02h	PCL	82h	PCL	102h	PCL	182h
STATUS	03h	STATUS	83h	STATUS	103h	STATUS	183h
FSR	04h	FSL	84h	FSR	104h	FSR	184h
PORTA	05h	TRISA	85h		105h		185h
PORTB	06h	TRISB	86h	PORTB	106h	TRISB	186h
PORTC	07h	TRISC	87h		107h		187h
PORTD①	08h	TRISD①	88h		108h		188h
PORTE①	09h	TRISE①	89h		109h		189h
PCLATH	0Ah	PCLATH	8Ah	PCLATH	10Ah	PCLATH	18Ah
INTCON	0Bh	INTCON	8Bh	INTCON	10Bh	INTCON	18Bh
PIR1	0Ch	PIE1	8Ch	EEDATA	10Ch	EECON1	18Ch
PIR2	0Dh	PIE2	8Dh	EEADR	10Dh	EECON2	18Dh
TMR1L	0Eh	PCON	8Eh	EEDATH	10Eh	Reserved②	18Eh
TMR1H	0Fh		8Fh	EEADRH	10Fh	Reserved②	18Fh
T1CON	10h		90h		110h		190h
TMR2	11h	SSPCON2	91h				
T2CON	12h	PR2	92h				
SSPBUF	13h	SSPADD	93h				
SSPCON	14h	SSPSTAT	94h				
CCPR1L	15h		95h				
CCPR1H	16h		96h				
CCP1CON	17h		97h				
RCSTA	18h	TXSTA	98h				
TXREG	19h	SPBRG	99h				
RCREG	1Ah		9Ah				
CCPR2L	1Bh		9Bh				
CCPR2H	1Ch	CMCON	9Ch				
CCP2CON	1Dh	CVRCON	9Dh				
ADRESH	1Eh	ADRESL	9Eh				
ADCON0	1Fh	ADCON1	9Fh				
通用寄存器 96字节	20h	通用寄存器 96字节	A0h	存取地址 20~7Fh	120h	存取地址 A0~FFh	1A0h
					16Fh		1EFh
					170h		1F0h
	7Fh		FFh		17Fh		1FFh

■ 未实现的数据存储器单元，读为“0”

*不是一个物理寄存器

① 这些寄存器未在PIC16F873A中实现

② 这些寄存器是保留的：保持这些寄存器为全0

图 7-6 PIC16F873A/874A 寄存器文件映射

7.3.3 配置字

PIC16 系列微控制器的配置字确定了微控制器的一些可编程特性。只能在对设备进行编程时来修改配置字;当器件工作时,不能对配置字进行修改。图 7-7 总结了 16F87XA 的配置字,它揭示了微控制器的一些基本特性。在图 2-6 中所示的 16F84A 的配置字中,我们已经对配置字的低 4 位和最高位比较熟悉了。下面来介绍一下其他位。两个"新"的操作模式是电路内编程和电路内调试,它们可以通过修改配置字被启用。一种很新且灵活的代码保护特性和局部电压过低检测(欠压)可通过 **BOREN** 配置位来启用。这些新增加的配置位会在后续部分详细阐述。

153~154

R/P-1	U-0	R/P-1	R/P-1	R/P-1	R/P-1	R/P-1	R/P-1	U-0	U-0	R/P-1	R/P-1	R/P-1	R/P-1
CP	—	DEBUG	WRT1	WRT0	CPD	LVP	BOREN	—	—	$\overline{\text{PWRTEN}}$	WDTEN	Fosc1	Fosc0

位 13 位 0

位 13 **CP**:Flash程序存储器代码保护位

位 12 未实现:读作"1"

位 11 **DEBUG**:电路内调试模式位

位 10和位9 **WRT1,WRT0**:Flash程序存储器写启用位

这两位的组合确定了在程序执行中程序存储器的哪个部分可以被写

WRT1:WRT0	PIC16F876A/877A		PIC16F873A/874A	
	被写保护的区域	可写的区域	被写保护的区域	可写的区域
11	无	所有	无	所有
10	0000h~00FFh	0100h~1FFFh	0000h~00FFh	0100h~0FFFh
01	0000h~07FFh	0800h~1FFFh	0000h~03FFh	0400h~0FFFh
00	0000h~0FFFh	1000h~1FFFh	0000h~07FFh	0800h~0FFFh

位 8 **CPD**:EEPROM数据存储器代码保护位

位 7 **LVP**:低电压(单电源)电路内串行编程启用位

位 6 **BOREN**:欠压复位启用位

位 5和位4 未实现:读作"1"

位 3 **$\overline{\text{PWRTEN}}$**:上电延时定时器启用位

位 2 **WDTEN**:看门狗定时器启用位

位 1和位0 $F_{OSC}1$:$F_{OSC}0$:振荡器选择位。11=RC,10=HS,01=XT,00=LP

注:所有位擦除后值为1

图 7-7 PIC16F87XA 配置字

7.4 "特殊"的存储器操作

传统的微控制器是从非易失性的程序存储器中读取指令,而使用易失性 RAM 来存放临时数据的。随着 Flash 存储器的引入,这 2 种传统存储器用法的区别将会逐渐缩小。Flash 存储器工艺是非易失性的,因此可以被用作程序存储器。因为可以很容

易地被改写，所以 Flash 存储器也可用作其他形式的存储器。

凭借 Flash 存储器工艺，16F87XA 微控制器允许运行中的程序在一些特定限制条件下修改程序存储器。它也允许对程序存储器进行串行编程(此时，微控制器已经被固定在目标应用系统中)。通过特殊的控制寄存器，器件允许程序存取 EEPROM 数据存储器。本节将会涵盖这些特殊的存储器功能。

7.4.1 存取 EEPROM 和程序存储器

通过设置配置字的 **WRT1** 和 **WRT0** 位，可以设置程序是否可修改程序存储器，这在上一节已经提到过。回顾图 7-7，可以发现这 2 位的不同设置可以允许程序修改程序存储器的不同块。

存取程序存储器单元需要通过数据和地址寄存器，它们与存取 EEPROM 是相同的(**EEDATA** 和 **EEADR**)，这在 2.4 节中已经阐述过。这些寄存器都是 8 位的，但是程序存储器中的数据是 14 位的。而且存取程序存储器时，需要 13 位的地址来访问 8K 字的空间(在 16F876A 和'877A 中)或 12 位来访问 4K 字的空间(在 16F873 和 874A 中)。因此，每个 **EEDATA** 和 **EEADR** 增加了一个“高字节”寄存器。这 2 个新加的寄存器称为 **EEDATAH** 和 **EEADRH**，它们只在访问程序存储器时才会用到。

图 7-8 描述了这种改进，图的上半部分是程序存储器，下半部分是 EEPROM。由 **EEADRH** 和 **EEADR** 组成的寄存器对用来寻址程序存储器，但是只有 **EEADR** 用来寻址 EEPROM。对于数据总线也是类似的，存取程序存储器时，**EEDATA** 和 **EEDATAH** 都携带数据；存取 EEPROM 时，只有 **EEDATA** 携带数据。

如果程序存储器的某个部分设置为可修改，那么 4 字的块必须同时被写。参考文献 7.1 对这个过程有详细的描述。但是单字的读是可以的。

控制寄存器 **EECON1** 控制所有的数据传输，如图 7-9 所示。首先注意到，寄存器的最高位用于设置 EEPROM 和程序存储器中的哪一个可以被修改。其他的位同图 2-7 中类似。一个主要的区别是中断标志位 **EEIF** 被转移到寄存器 **PIR2** 中(如图 7-13所示)。

存取 EEPROM 和程序存储器并不简单，必须仔细按照特定的操作顺序。写这 2 个存储器时，必须按顺序将 55h 和 AAh 写入 **EECON2** 寄存器中。这种操作增加了写过程的安全性，确保不会有意外的写操作发生。参考文献 7.1 给出了写操作代码的实例。这种操作顺序可以根据具体情况做适当的调整。

如果配置字的 **CP** 位设置为 0，那么从微控制器的外部将不能读取程序存储器的内容，比如使用 PICSTART® ＋编程器。这种设置是为了保护设计者或者生产者的知识产权。但是它不会影响微控制器内部对程序存储器的访问。类似地，对 EEPROM 的存取是由配置字的 **CPD** 位控制的。

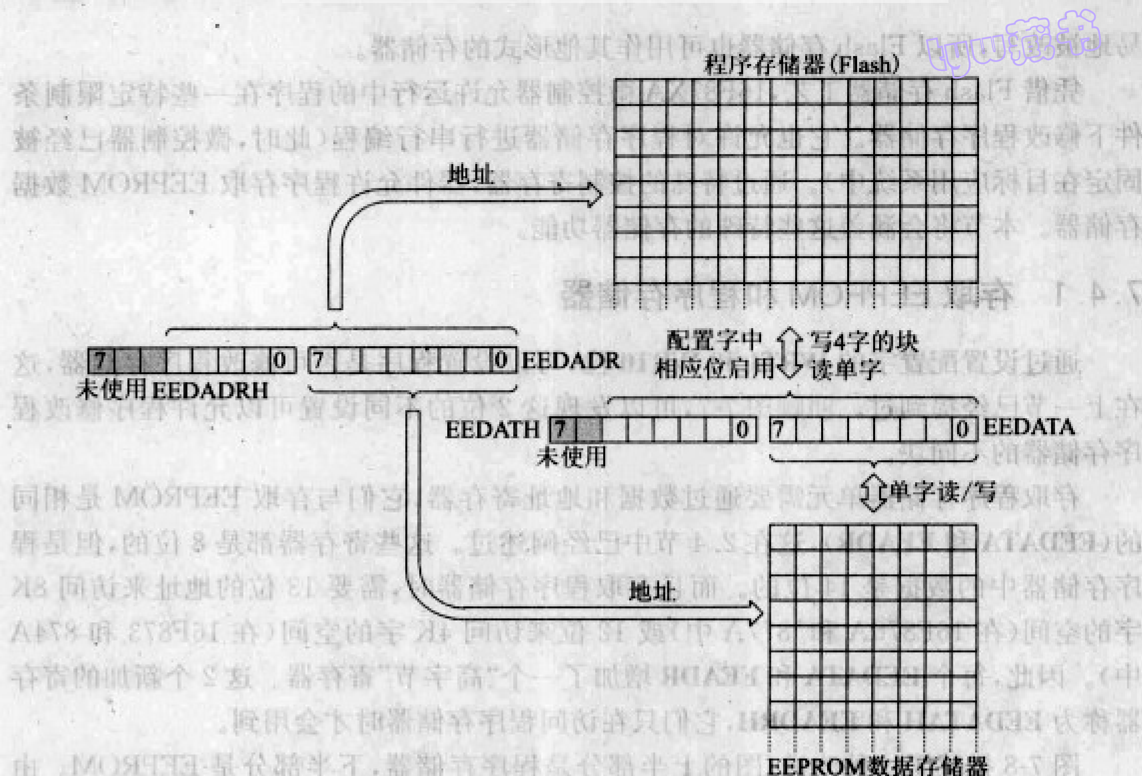


图 7-8 写程序存储器和 EEPROM

R/W-x	U-0	U-0	U-0	R/W-x	R/W-0	R/S-0	R/S-0
EEPGD	—	—	—	WRERR	WREN	WR	RD
位 7							位 0

位 7 **EEPGD**：程序/数据EEPROM选择位
1=存取程序存储器
0=存取数据存储器
POR(上电复位)复位后值为"0"，当一个写操作正在进行时，不能改变这一位

位 6~4 未实现：读为'0'

位 3 **WRERR**：EEPROM错误标志位
1=写操作过早中止(在正常操作下，每次$\overline{\text{MCLR}}$复位或WDT复位时)
0=写操作完成

位 2 **WREN**：EEPROM写启用位
1=允许写周期
0=禁止写EEPROM

位 1 **WR**：写控制位
1=开始写周期。一旦写完成，WR通过硬件清零。WR位只能在软件中设置(不能清零)
0=EEPROM写周期完成

位 0 **RD**：读控制位
1=开始读EEPROM。RD通过硬件清零。RD位只能在软件中设置(不能清零)
0=未开始读EEPROM

图 7-9 EECON1 寄存器

7.4.2 电路内串行编程（ICSP™）

所有16F87XA系列的微控制器都可以在最终应用电路中进行串行编程。这种功能非常重要。在开发环境中，电路内编程允许测试程序直接下载到目标应用系统中的微控制器，而不需要将微控制器从系统中取出来放在编程器中进行。在生产环节，它允许产品在出厂之前，对嵌在其中的微控制器进行现场编程。因此，可以在产品中安装最新的软件。当产品交付使用之后，电路内编程还能允许程序通过网络进行更新，而产品的使用者甚至不知道程序被更新。 156

电路内编程也有一些缺点。一些引脚必须具备编程功能，或者精心地设计引脚使它具有双重功能：正常功能和编程功能，但是必须要保证正常功能不能妨碍编程功能。电路内编程（ICSP）使用的引脚如表7-1所示。在Derbot AGV项目中，我们会使用电路内编程。

在正常的编程模式下，需要加一个13V左右的电压到**MCLR**引脚。但是，电路内编程是一种特殊模式，并不需要这么高的电压。它是一种低电压编程模式（Low-Voltage Programming，LVP）。这种模式可以通过配置字的**LVP**位启用，它允许在正常电压V_{DD}下进行编程。它的缺点就是引脚RB3必须用作编程功能，不再作为普通的I/O引脚，如表7-1所示，该引脚用来控制进入和退出低电压编程模式。

除了硬件引脚的要求外，电路内串行编程还需要一些非常特殊的程序用于将数据串行地传输到微控制器。这里不再赘述，可查阅参考文献7.2。 157

表7-1 电路内串行编程使用的引脚定义

引脚名称	描　述
RB3	低电压编程模式中的控制输入（配置字中的LVP位为1）
RB6	时钟输入
RB7	数据输入/输出
$\overline{MCLR}$	编程模式选择，编程电压连接
V_{DD}	电源
V_{SS}	接地

7.5 16F873A的中断

7.5.1 中断结构

作为PIC 16系列的一个成员，16F873A的结构应该同该系列中的其他微控制器的结构类似。这种设计策略在某些地方很适用，但是对于PIC 16系列微控制器的某些结构，我们会发现这种设计策略的局限性。16F873A的中断结构就是一个例子，如 158

图 7-10所示。从图中我们看到图 6-2 中所示的 16F84A 的最低要求的中断结构被修改成 16F873A 的中断结构。但是当前图中的 EEPROM 写完成中断被一条至少连接 11 个中断源，跨越整个微处理器的线所替代。所有这些中断最后都汇集成一个中断向量，如图 7-4 所示的程序存储器映射结构中的中断向量。

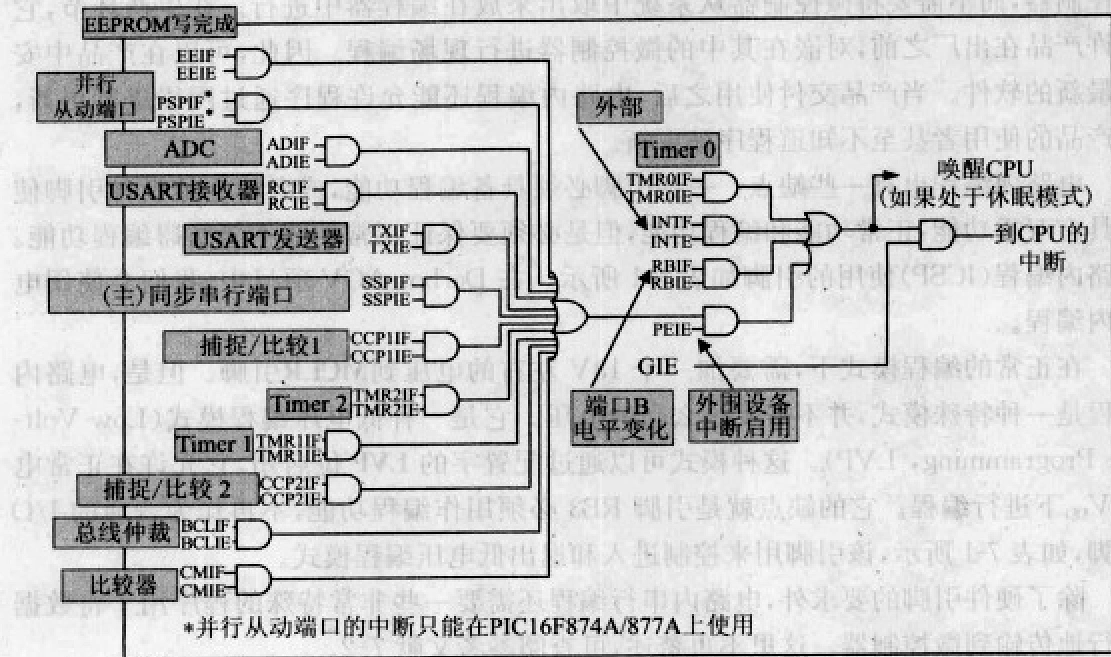


图 7-10 PIC16F87XA 中断结构(阴影框中所附标签为作者所加)

7.5.2 中断寄存器

上一章已经介绍了规模较小的 PIC 16 系列微控制器的中断寄存器 **INTCON**(如图 6-3所示)。由于 PIC 16F873A 有 15 个中断源，**INTCON** 只能保存其中一部分的中断标志和启用信息。因此，对 16F87XA 的 **INTCON** 寄存器做了一些改动，如图 7-11 所示。除了 **EEIE**(EEPROM 写完成中断启用)被 **PEIE** 取代之外，**INTCON** 寄存器基本保持不变，如图 7-10 所示。**PEIE** 是外围设备中断启用位(Peripheral Interrupt Enable bit)。对于所有的外围设备中断源来说，它作为辅助的全局中断启用。如果需要允许任何一个外围设备中断，必须将 **PEIE** 置为 1。

159

由于 **INTCON** 寄存器不能保存所有的中断信息，因此增加了 4 个新的特殊功能寄存器——**PIE1**、**PIE2**、**PIR1** 和 **PIR2**。**PIE1** 和 **PIE2** 用于保存中断启用，在存储区 1 中；**PIR1** 和 **PIR2** 用于保存中断标志，在存储区 0 中。这 4 个寄存器如图 7-12 和图 7-13 所示。**PIE1** 和 **PIR1** 具有相同的格式，**PIE2** 和 **PIR2** 也具有相同的格式。这 4 个寄存器中的有效位在上电时被置为 0。

160

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF
位7							位0

位7 **GIE**：全局中断启用位
1=启用所有未屏蔽的中断
0=禁止所有中断

位6 **PEIE**：外围设备中断启用位
1=启用所有未屏蔽的外围设备中断
0=禁止所有外围设备中断

位5 **TMR0IE**：TMR0溢出中断启用位
1=启用TMR0溢出中断
0=禁止TMR0溢出中断

位4 **INTE**：RB0/INT外部中断启用位
1=启用RB0/INT外部中断
0=禁止RB0/INT外部中断

位3 **RBIE**：RB端口电平变化中断启用位
1=启用RB端口电平变化中断
0=禁止RB端口电平变化中断

位2 **TMR0IF**：TMR0溢出中断标志位
1=TMR0寄存器已经溢出(必须在软件中清零)
0=TMR0寄存器尚未发生溢出

位1 **INTF**：RB0/INT外部中断标志位
1=发生RB0/INT外部中断(必须在软件中清零)
0=未发生RB0/INT外部中断

位0 **RBIF**：RB端口电平变化中断标志位
1=RB7：RB4引脚中至少有一位的状态发生了变化：不恰当的状态会接着将该位置位。读PORTB会结束不恰当的状态，并且允许该位被清零(必须在软件中清零)
0=RB7：RB4引脚没有改变状态

图 7-11 16F87XA **INTCON** 寄存器

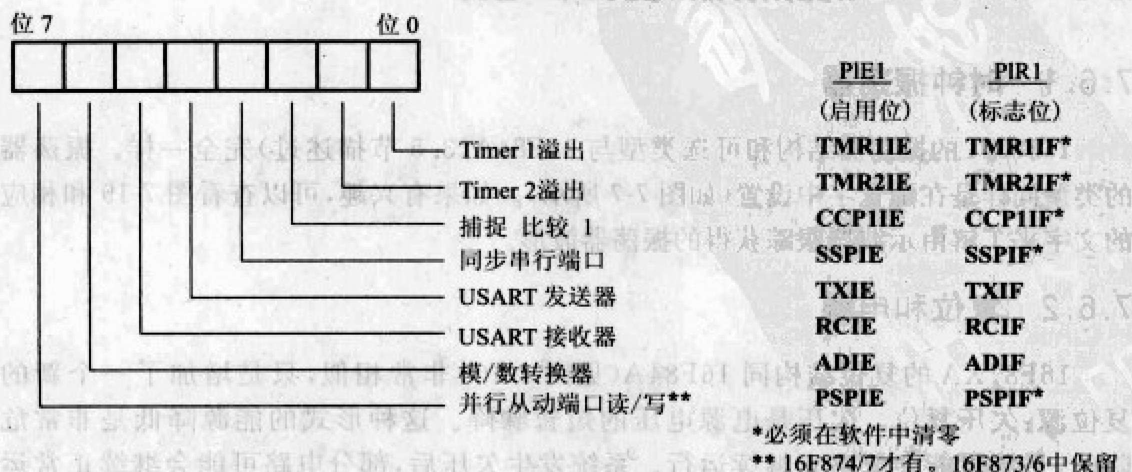


图 7-12 16F87XA **PIE1/PIR1**(外围设备中断启用/外围设备中断请求)寄存器

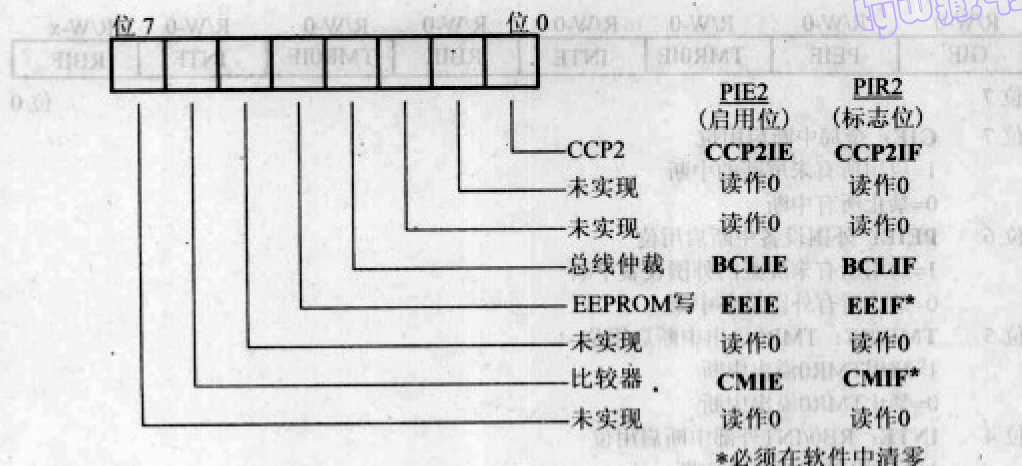


图 7-13 16F87XA **PIE2/PIR2** 寄存器

7.5.3 中断识别和上下文保存

如果有多个中断源被启用,需要在中断服务程序(ISR)起始处设置一段程序用来识别当前触发的中断类型。这与 16F84A 的情况非常相似,因此可以使用例程 6-2 的方法来识别触发的中断类型。

16F87XA 的上下文保存与 16F84A 是一样的——仅将中断返回地址压入栈即可。由程序员在中断服务程序开始处保存中断服务程序需要用到的其他寄存器,它们通常是 W 寄存器和状态寄存器。在中断服务程序结束时,可以采用例程 6-4 的方法,再恢复这些寄存器的值。

7.6 16F873A 的振荡器、复位和电源

7.6.1 时钟振荡器

1687XA 的振荡器结构和可选类型与 16F84A(3.5 节描述过)完全一样。振荡器的类型同样是在配置字中设置(如图 7-7 所示)。如果有兴趣,可以查看图 7-19 和相应的文字来了解由示波器跟踪获得的振荡器波形。

7.6.2 复位和电源

16F87XA 的复位结构同 16F84A(见图 2-10)非常相似,只是增加了一个新的复位源:欠压复位。欠压是电源电压的短暂骤降。这种形式的能源降低是非常危险的,系统可能会忽略它继续运行。系统发生欠压后,部分电路可能会继续正常运行,但是另外一些电路可能会暂时失效或者丢失数据。系统中增加了欠压复位逻辑之后,可以保证设备在发生欠压时,整个设备都进入复位状态。配置字的

BOREN 位可以禁用或启用欠压复位电路。如果 BOREN 位置高，当发生欠压时，微控制器被强制复位。设备参数[7.1]中给出了 4V 的典型电压；低于该电压时将会触发欠压复位。除此之外，1687XA 和 16F84A 具有几乎相同的电源要求，如图 3-16 所示。不同之处在于当欠压复位被启用，最小的电源电压由欠压复位（Brown-Out Reset）来确定。

现在有多个复位源可用，这就需要由程序来辨别最近一次发生了何种类型的复位。状态寄存器的 $\overline{TO}$ 位用来指示是否发生了看门狗复位（图 7-3）。PCON 寄存器的 $\overline{POR}$ 和 $\overline{BOR}$ 位提供了更多的复位信息。当发生上电复位时，$\overline{POR}$ 位被置为 0；当发生欠压复位时，$\overline{BOR}$ 位被置为 0。

7.7 16F873A 的并行端口

如图 7-2 所示，PIC16F873A 有 3 个端口：A、B 和 C。端口 A 和 B 与 16F84A 类似，只是在引脚上赋予更多的功能，端口 A 由 5 位扩展为 6 位。所有的端口在复位后设置为输入。电源为 5V 时，端口的输出特性如图 7-14 所示。应用 3.4.3 节的公式可以推导出引脚逻辑电平为 1 时，典型的输出阻抗约为 70Ω；逻辑电平为 0 时，输出阻抗约为 22Ω。 161

下面依次研究一下端口 A、B、C 的特性。

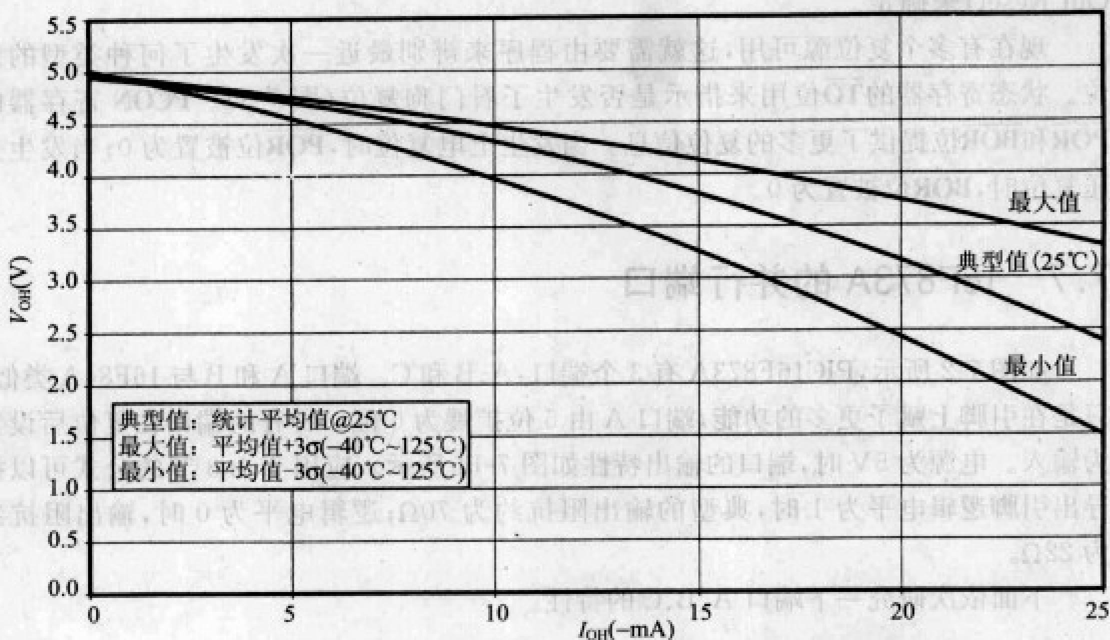
7.7.1 16F873A 的端口 A

如图 7-1 所示，端口 A 的 6 位连接到芯片的 2～7 号引脚上。查看一下引脚上的关键词，了解引脚的连接是很有必要的。端口 A 可以作为一个通用双向数字端口。同时，它还具有模拟功能，特别是在模/数转换器（ADC）模块和比较器中。值得注意的是，上电时端口被设置为模拟输入。在第 11 章我们会学到这 2 种功能。如果要使用端口的数字功能，必须正确地设置 ADCON0 寄存器，这在第 11 章会提到。同 16F84A 一样，重要的外围设备部件 Timer 0 的输入在端口 A 的 4 号引脚上。

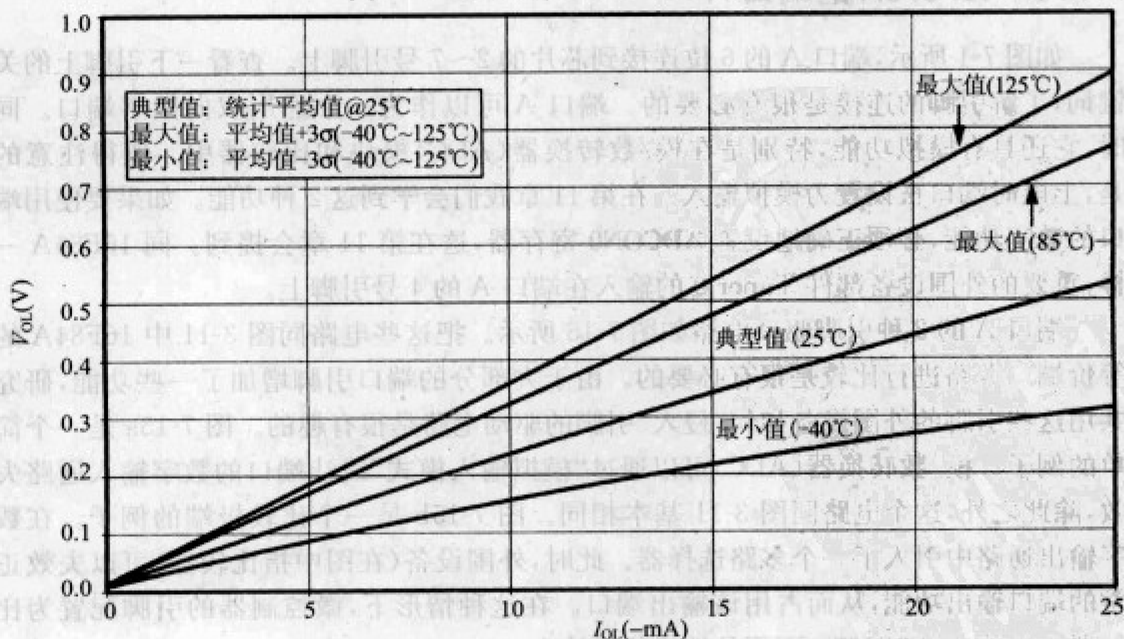
端口 A 的 2 种引脚驱动电路如图 7-15 所示。把这些电路同图 3-11 中 16F84A 的等价端口电路进行比较是很有必要的。由于大部分的端口引脚增加了一些功能，研究共用这些引脚的外围设备如何“侵入”引脚的驱动电路是很有趣的。图 7-15a 是一个简单的例子。模/数转换器（ADC）可以通过“模拟输入模式”线让端口的数字输入通路失效，除此之外，这个电路同图 3-11 基本相同。图 7-15b 是一个比较极端的例子。在数字输出通路中引入了一个多路选择器。此时，外围设备（在图中指比较器）可以失效正常的端口输出功能，从而占用该输出端口。在这种情形下，微控制器的引脚配置为比较器 2 的输出（C2OUT），而不是端口位的输出。 162～163

引脚 4 的驱动电路在图中没有画出。它基于图 3-11b 的电路，只是在输出通路上增加了比较器 1 需要的多路选择器，同图 7-15b 中的电路类似。

研究这些引脚驱动电路，我们可以得出一个重要的结论：端口引脚不再简单地受端口寄存器"TRIS"的控制，其他的 SFR 也可以失效或者重新分配引脚资源。这会导致较大的编程困难：由于外围设备的 SFR 可能会错误地重新分配了引脚的功能，导致引脚无法正常工作。因此，程序员必须十分谨慎。



(a) 典型 V_{OH} 和 I_{OH}(V_{DD}=5V, −40℃~125℃)关系曲线



(b) 典型 V_{OL} I_{OL}(V_{DD}=5V, −40℃~125℃)关系曲线

图 7-14 PIC16F87XA 端口的输出特性

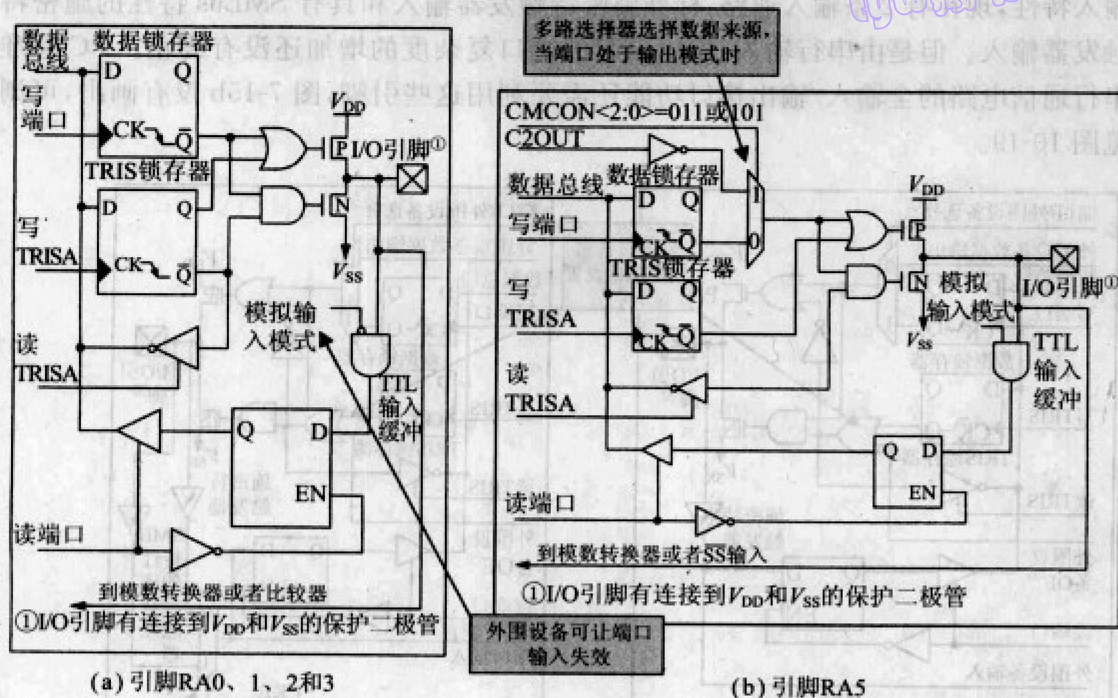


图 7-15 PIC16F87XA 端口的输入特性(阴影框中所附标签为作者所加)

7.7.2 16F873A 的端口 B

如图 7-1 所示，端口 B 的 8 位连接到芯片的 21～28 号引脚上。它与 16F84A 的端口 B 几乎完全相同，继续保持单功能端口。端口驱动电路在本质上也同 16F84A 相同，如图 3-11 所示。图 7-1 和表 7-1 显示，位 3、位 6、位 7 用于电路内串行编程(In-Circuit Serial Programming，**ICSP**)。当使用电路内串行编程功能时，设计者必须保证：要么不将这些引脚用于其他功能，要么当引脚用于其他功能的同时，它们仍然可以用于电路内串行编程。

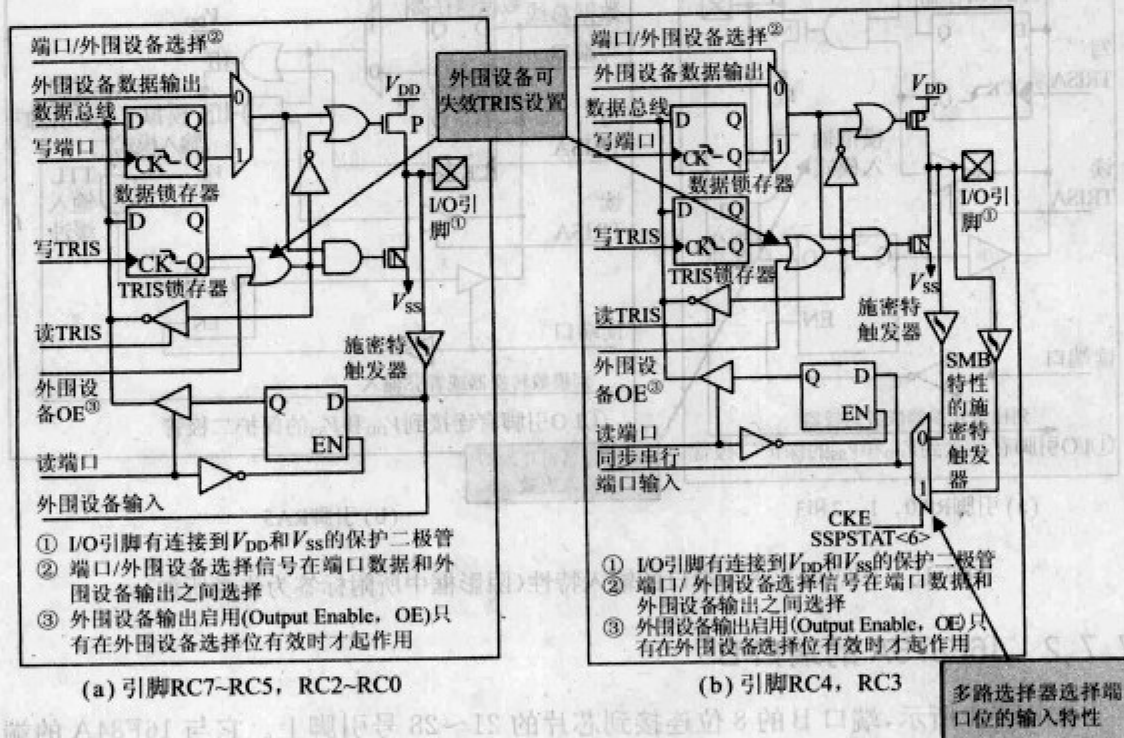
7.7.3 16F873A 的端口 C

如图 7-1 所示，端口 C 的 8 位连接到芯片的 11～18 号引脚上。它是 16F873A 中最为复杂的端口。它的引脚可作为通用双向数字输入/输出(I/O)端口。有趣的是所有的引脚输入都有施密特触发特性。除了用作通用数字输入/输出外，端口 C 的引脚被一些更复杂的微控制器外围设备共用，比如处理串行通信的外围设备。

图 7-16 为端口 C 的引脚驱动电路。从图中我们可以发现，它们也与简单的 16F84A 的引脚驱动一样，可用作通用数字输入/输出端口。另外，它在端口的输出通路上增加了多路选择器，同端口 A 类似。此外，通过“外围设备输出启用”线和其驱动的或门，外围设备可以控制 **TRIS** 的功能，如图中所示。对标准端口引脚电路的最终改进如图 7-16 所示。由于系统管理总线(system management bus，SMBus)改变了端口的

输入特性，现在有 2 条输入通路：标准施密特触发器输入和具有 SMBus 特性的施密特触发器输入。但是由串行输入/输出引起的端口复杂度的增加还没有终止。I²C 标准串行通信电路的全输入/输出接口功能还需要利用这些引脚，图 7-15b 没有画出，可浏览图 10-19。

164



(a) 引脚RC7~RC5，RC2~RC0　　(b) 引脚RC4，RC3

图 7-16　端口 C 的引脚驱动电路框图(阴影框中所附标签为作者所加)

7.8 测试、调试、诊断工具

我们现在学习的是一个更复杂的微控制器系列，因此我们会在此基础之上开发一个更加复杂的系统。所以我们有必要对如何测试和调试这样的系统进行思考。下面几节会阐述进行测试和诊断时需要使用的设备和技术，在开发 Derbot AGV 项目时我们也会使用到它们。

7.8.1 测试嵌入式系统的挑战

4.1.3 节概述了嵌入式软件的开发过程，其中包含了软件仿真器的使用。一旦程序下载到硬件上实际运行，问题会大大增加，特别是当软硬件都没有经过测试时。此时，开发者可能会经历一段非常痛苦的调试过程，这也是所有嵌入式系统开发者都会遇到的：系统完全无法运行，但是又不知道问题出在何处。

165

在测试过程中，开发者主要会遇到下面两种类型的问题。

□ 设计问题。这是软硬件设计的问题，它们会导致部分或者整个系统无法工作。

□ 实现问题。这类问题是由设计的实现方式引起的，设计可能是完全正确的。比如 PCB 电路板坏掉、不成功的或者错误的程序下载、下载过程中不正确的配置字设置。这些问题同样会导致部分或者整个系统无法工作。

不同类型的错误可能会出现同样的现象。辨别错误是哪种类型非常重要，因为针对不同类型的错误解决方法也是完全不同的。

下面试图建立一个调试层次，帮助我们进行系统性的测试和调试。图 7-17 使用非正式的方式表述了嵌入式系统不同调试阶段之间存在的依赖层次。一般地，测试过程是从图底端开始向上进行。如果一辆汽车没有汽油，它无法行驶。类似地，如果没有给微控制器提供合适的电源，它也无法运行，尽管一些质量高的电路或者程序可能会照常运行。同样地，如果振荡器没有振荡，复位引脚没有起作用（$\overline{\textbf{MCLR}}$为 PIC 微控制器的复位引脚），或者程序没有正确下载，电路都将无法正常运行。一旦满足了这 4 个必要的条件，程序才有可能开始运行。*如果一个系统完全没有任何反应，首先需要检查这些条件*。出现这种现象，问题很有可能属于实现错误，比如晶体同电路板的焊接出现问题或者程序由于某种原因被错误下载。

166

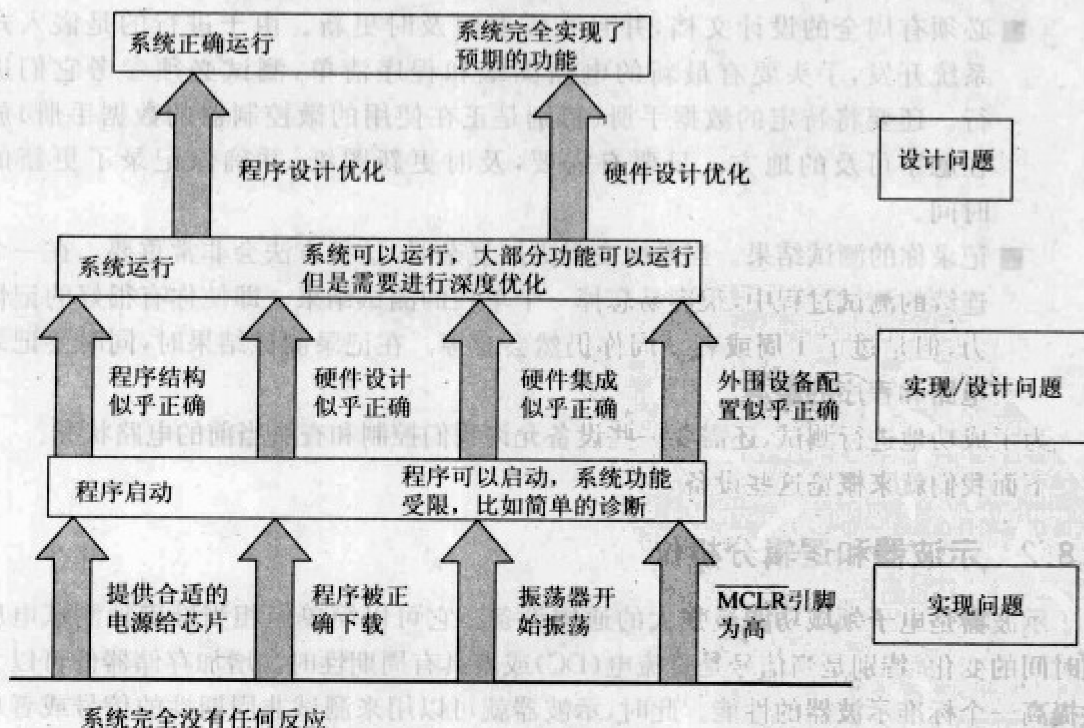


图 7-17　嵌入式系统的调试依赖层次

一旦满足了这些基本的条件之后，当系统可以持续运行并完成了适中的功能要求时，可以应用进一步的规则。这些规则包括：似乎正确的电路和程序、正确的硬件集成、所有的外围设备似乎被正确配置。这里似乎正确一词表示设计没有重大致命的问

题，但是需要对某些地方进行改进。当上述的这些条件都满足时，系统开始进入优化阶段。在这个阶段，系统的功能都已经实现，只是某些设计还是不太完美。从该阶段开始，可以利用正在进行的增量设计和开发对系统的硬件或者软件进行更进一步测试。最后，系统完全实现了预期的功能。

假设已经理解了嵌入式系统的这种调试依赖层次，我们如何去执行这个测试过程，特别是一旦图 7-17 底端层次的测试已经完成时？在参考文献 1.1 中，文献作者总结了测试的 3 条"黄金规则"。

- 分割。将软硬件系统划分成一些尽量独立的子模块或者子系统，单独对它们进行测试。然后系统再从"已知正确"的子系统中重新建立起来。
- 当设计通过验证之前，假定它们是错误的。在你没有验明电路是正确之前，不要对系统任何部分进行正确性的假设。即在你没有验证之前，假定它无法正常运行。这同法律正好相反：在证明有罪之前，被告假定无罪。
- 测试系统化，记录严格化。规则可扩展为以下几点。
  - 有计划地进行测试，保证测试过程是理智和逻辑的。图 7-17 在某种程度上指出了测试开始的起点。测试从图的最底端开始，依次向上进行。
  - 必须有周全的设计文档，并且在需要时及时更新。由于进行的是嵌入式系统开发，手头要有最新的电路图纸和程序清单，测试必须参考它们进行。还要将特定的数据手册（特别是正在使用的微控制器的数据手册）放在触手可及的地方。只要有需要，及时更新图纸，并确保记录了更新的时间。
  - 记录你的测试结果。当系统变得更加复杂时，这种方法会非常重要。在一个连续的测试过程中，很容易忘掉一个单独的测试结果。即使你有很好的记忆力，但是过了 1 周或者 2 周你仍然会忘掉。在记录测试结果时，同时要记录电路和程序的版本。

为了成功地进行测试，还需要一些设备允许我们控制和查看当前的电路状况。下面我们就来概览这些设备。

7.8.2 示波器和逻辑分析仪

示波器是电子领域功能最强大的通用设备。它可以简单但相当准确地测试电压随时间的变化，特别是当信号是直流电(DC)或者具有周期性时。增加存储器件可以大大提高一个标准示波器的性能。此时，示波器就可以用来测试非周期性的信号或者单独的事件。在电子领域，示波器是用来对电路的基本条件，如电源、晶振、简单的端口活动进行测试的设备。一个专家级的示波器，可以用来查看更复杂的信号。

逻辑分析仪具有示波器的一些特性，因为它也可以随着时间显示一个输入信号的值。但是输入信号被认为是数字的，其值要么是 0，要么是 1。电压的阈值由逻辑分析仪确定，对于大部分设备来说，该值可以调整。逻辑分析仪的一个常规特性是它具有

多个输入，比如16个、32个或者48个。因此，它可以用来查看数据和地址总线的活动。逻辑分析仪通常具有一些额外的特性，如存储器可以存储输入信号的历史变化以及复杂的触发能力——可以指定需要识别的输入信号组合，识别后将引发一个触发。以前，系统由多个IC构建起来时，逻辑分析仪非常流行，因为需要查看总线的活动状况。现在的许多系统中，IC非常复杂，总线的接口不再容易接触到。但是在查看复杂的数字信号变化方面，比如并行端口或者串行数据流，它还是非常有用的。

本书使用了一个很出色的设备，它集成了示波器和逻辑分析仪2者的特点。它就是Agilent 54622D混合信号示波器，图7-18展示了它的外形。它有2个传统的示波器输入和16个逻辑分析仪输入。任何输入信号的都能立即显示出来。这台“示波器”的触发机制、信号分析和存储功能都极为强大。它的模拟和数字信号混合分析功能反映了大部分嵌入式系统是模拟和数字的混合设计。在嵌入式开发环境中，它很适合对系统进行全面的测试。

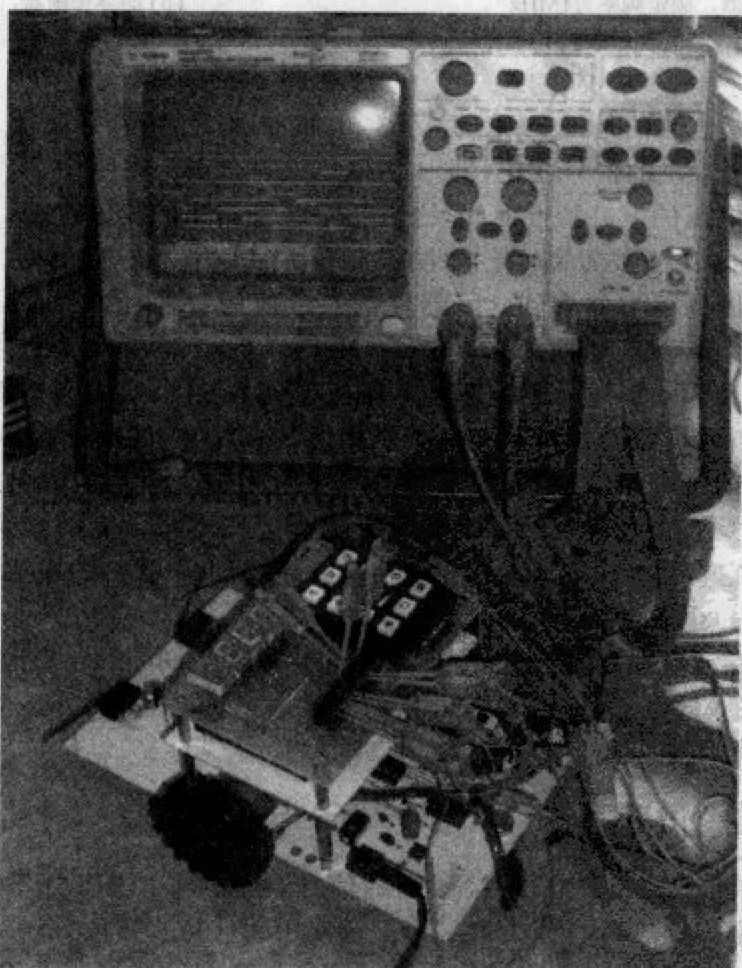
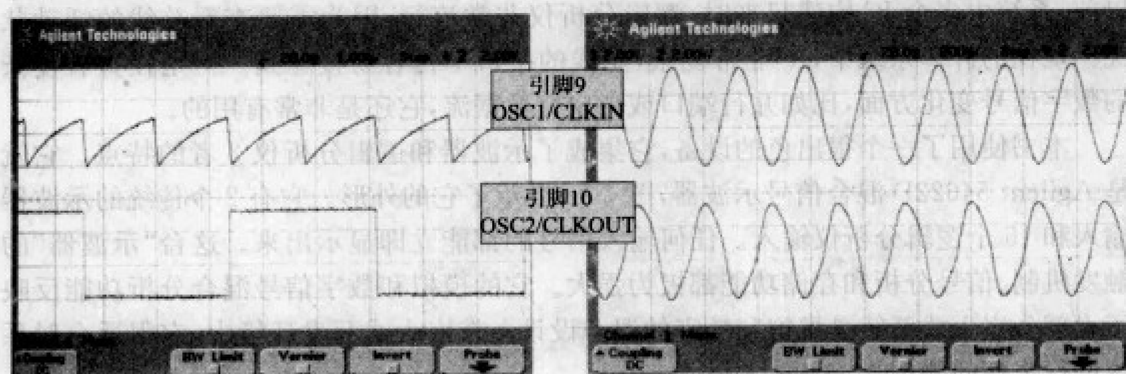


图7-18 连接到Derbot AGV的Agilent 54622D混合信号示波器

使用 Agilent 示波器的模拟输入得到的两张屏幕截图如图 7-19。注意屏幕上端的通道 1 和通道 2 的垂直刻度为 2V/cm，水平刻度为 1μs/cm。屏幕左边的 2 个小箭头指出了跟踪信号的 0 电平参考位置。



(a) RC振荡器，额定频率为1MHz　　(b) 晶体振荡器，4MHz

图 7-19　19F873A 设备的振荡器波形

第 1 个屏幕监测的是一个 RC 振荡器，电路如图 3-14b 所示。上半部描绘的是微控制器 OSC1 端点的信号变化，即 R_{EXT}和 C_{EXT}的交点。在图中，可以看到电容通过电阻充电时电压的上升特性曲线，之后紧跟着是迅速的放电过程，这是由于施密特触发器的输出开关被置高了。如图 3-14b 所示，$F_{OSC}/4$ 为 OSC2 引脚的信号。这可以通过左图的下半部看到。元器件的参数值为：$R_{EXT}=8.2\text{k}\Omega$，$C_{EXT}=100\text{pF}$，晶振的频率为 700kHz(周期为 1.43μs)。可以看到，实际信号的周期非常接近于 1.4μs(刚刚超过 700 kHz)，$F_{OSC}/4$ 的周期为 5.6μs。这个结果同预料的非常吻合。应该注意到，当示波器的探针从 OSC1 处移走时，$F_{OSC}/4$ 的周期会跌至 5.0μs。这是使用示波器的一个缺陷：探针和示波器的输入附加了一个负载电容(和电阻)到测试电路中。是否会有较大的影响取决于测试电路本身。上面这个例子中，负载大约是 15pF，它直接与电阻 R_{EXT}并连，导致电路增加了 10%的电阻值。这是由测试本身引入的一个测试错误。

第 2 个跟踪波形是微控制器的 HS 晶振模式，电路如图 3-14a 所示。2 个微控制器引脚具有相似的、近似 sin 函数、频率相同的波形。一个信号是另外一个的取反。可以清楚地看到信号周期为 125ns，即频率为 4MHz。

168

7.8.3　电路内仿真器

当系统运行时，对于测试可接触到的信号，示波器和逻辑分析仪是非常有用的。但是，我们需要进行一些类似于第 5 章使用软件仿真器的那种方式的测试，比如测试特定段的代码或者单步执行程序。区别是现在的测试要求代码在目标硬件中运行。

可以满足这种要求的是电路内仿真器(in-circuit emulator，ICE)。这种设备可以在

电路中替代实际的微控制器。它尽量模仿微控制器的动作，但是必须与主机相连。主机拥有控制程序执行的能力，如同软件仿真器的执行方式。

电路内仿真器是对嵌入式系统进行测试的功能强大的方式之一。但是它有一些缺点。

- □它们由非常复杂的功能块组成，因此价格非常昂贵。
- □它们只能替代特定的微控制器或者微处理器，因此不同的微控制器需要不同的电路内仿真器。
- □有一些地方，电路内仿真器的仿真是不准确的。比如考虑到时钟振荡器时，它们并不能很好的仿真微控制器的行为，因为仿真器对电源的要求没有微控制器本身那么灵活。
- □它们不能模拟真实的操作条件。毕竟，微控制器被移除了。

7.8.4 片上调试器

当考虑了电路内仿真器的优缺点之后，集成电路的设计者会很自然地联想到是否可以将电路内仿真器的特性集成到微控制器中。毕竟，一个好的电路内仿真器本身就是微控制器电路的复制。因此，片上调试的概念开始形成了。不同的公司对它的命名也不同。Motorola公司（现为Freescale公司）使用术语*后台调试模式*（background debug mode，BDM），而Microchip公司使用术语*电路内调试器*（in-circuit debugger，ICD）。由于许多PIC微控制器都可以使用它，包括PIC16F873A，因此我们会仔细研究这个功能。

电路内调试器的**优点**总结如下所示。

- □测试不会影响目标系统的运行。
- □电路内调试器可以作为编程器使用，用它将程序下载到目标微控制器中。
- □电路内调试器的方法更加灵活。连接、测试、甚至程序下载都可以“在现场”进行，如同开发系统在实验室中一样。

电路内调试器的**缺点**总结如下所示。

- □一些微控制器的资源被电路内调试器占用，包括一些输入/输出端口、程序存储器和其他内部资源。
- □目标微控制器必须在时钟运行的条件下才能正常运行。
- □一般来说，它的功能弱于一个速度很快的ICE系统。
- □对一些硬件的测试，它的效果不是很好。比如当微控制器的程序被电路内调试器停止之后，外围设备可能还会继续运行，这将导致错误的结果。

169
~
170

7.9 Microchip 公司的电路内调试器(ICD 2)

Microchip 公司的 ICD 2 系统的特性如图 7-20 所示。主机通过一个特殊的适配器单元或者插座与目标系统连接。在图 7-21 中的真实系统中，ICD 2 插座与 Derbot AGV 相连。ICD 受主机控制，主机上需要运行 MPLAB® 程序。ICD 2 插座通过 USB 电缆线与主机连接。也可使用 RS232 连接方式进行连接，通过一个有 5 个接口的电缆线连接，如图 7-20 所示。这种连接方式与 ICSP 基本相同，只是没有使用低电压编程引脚(端口 B 的位 3)。电路内调试器需要的微控制器内部资源如图 7-20 所示，包括程序存储器、数据存储器和栈。

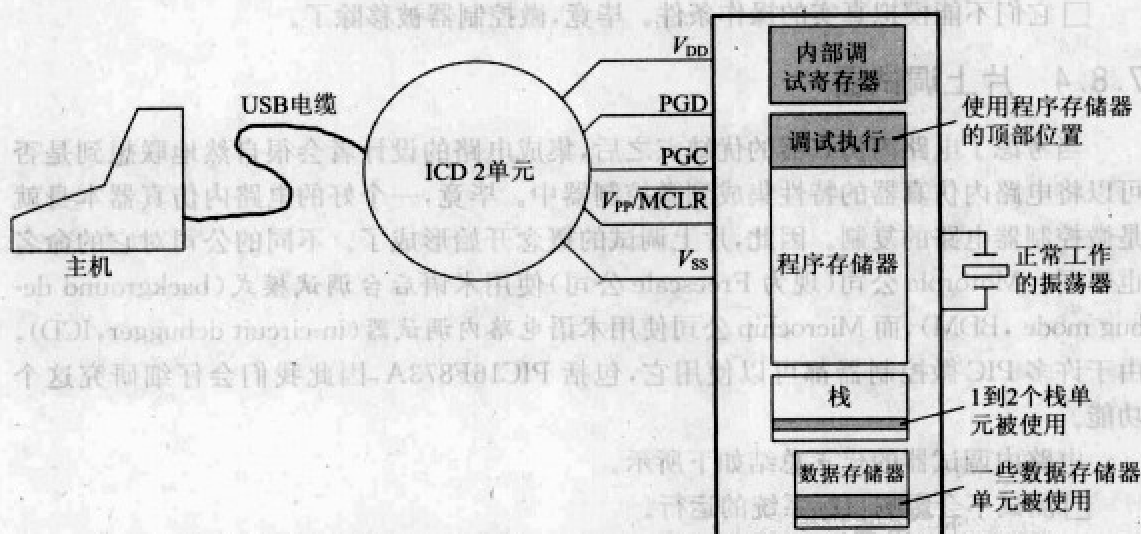


图 7-20 ICD 2 系统和内部资源需求

ICD 2 可作为调试器使用。在这种方式下，它可以对微控制器进行编程，同时将它自己运行需要的"调试程序"下载到程序存储器的顶部。在 MPLAB 程序上运行调试器，可以完成如同第 4 章和第 5 章介绍的 MPLAB 仿真器所有的功能，唯一的区别是现在的程序是在实际硬件上运行。ICD 2 也可作为编程器使用。在这种方式下，它执行与编程器相同的操作，例如第 4 章的编程器 PICSTART Plus。可以在 MPLAB 中来选择操作模式。

171 在产品的开发周期中，使用 ICD 2 进行程序调试的步骤如下。在调试模式下，使用 ICD 将程序下载到微控制器中，然后使用 ICD 的所有功能对程序进行调试。当程序可以完全运行之后，ICD 被转换到编程器模式。不携带调试器相关的特殊信息的程序再次被下载到微控制器中，此时微控制器可以在正常模式下运行。

在第一次构建 Derbot AGV 项目时，将会介绍基于自主导向车项目的 ICD 2 调试器的使用方法。



图 7-21 连接到 Derbot AGV 的 ICD2

7.10 应用 16F873A:Derbot AGV

现在我们开始研究 16F873A 微控制器在 AGV 中的应用。AGV 的框图如图 1-5 所示,图 A3-1 为其电路图。一些 AGV 在设计和应用方面相关的基本考虑因素在附录 4 中进行了描述。

7.10.1 电源、振荡器和复位

AGV 需要 2 组电源:一个用来给电机供电;一个用来给微控制器芯片和所有相关的传感器和显示器件供电。虽然可以利用一个低电压的电机(在 2V 或者 3V 的电压下工作),但是设计一个工作在此电压下的变换电路并非易事。通过一些电机的试验,可以获得一个在电流可忍受、电压为 9V 时,合适的发送机力矩和速度。这个电压非常适合当前可用的电机接口芯片,比如 Derbot AGV 中使用的 L293D。第 2 个电源是给微控制器芯片供电的。尽管微控制器芯片可以在非稳压电源下工作,如在电子乒乓球游戏项目中,但是有许多传感器、特殊的线性电阻和反射性的光学传感器需要稳定的电压。因此,必须选择稳压电源。

172

AGV 一般由 6 节"AA"碱性电池供电,可得到 9V 的额定电压。这个电压"直接"用来给电机供电。为了给微控制器和周边电路供电,使用国家半导体公司的 LP2950 的低电压变换芯片将 9V 电压变换到 6V。许多系统器件,比如光学传感器,依赖这个变压后的电压才能正常工作。

由于应用中有一些精确的定时要求,因此选择使用晶体振荡器。振荡器频率为 4MHz,那么定时程序可以利用对应的 1μs 指令周期进行定时。

7.10.2　并行端口的使用

173

下面会看到，AGV 的输入/输出需求非常大，因此所有的这些需求并不能完全在任何一个 AGV 版本中实现。表 7-2 列出了这些需求。有一些需求只是需要通用的并行输入/输出端口。另外一些则属于特殊功能，通过单独指定的引脚将它们连接到特定的外围设备或者输入/输出端口。因此这些功能被直接分配到它们的特殊功能引脚上。像端口引脚本身，在表中有一些共用的功能，这样做主要是为了简化表述。

表 7-2　AGV 输入/输出需求

功　能	数据方向	使用的微控制器外围设备	端口和引脚分配
电路内调试	双向	并行端口	端口 B，引脚 6 和引脚 7
“碰撞”微开关	输入	并行端口	端口 B，引脚 4 和引脚 5
伺服电机驱动	输出	并行端口	端口 B，引脚 3
光学传感器启用	输出	并行端口	端口 B，引脚 2
压电式发声器	输出	并行端口	端口 B，引脚 1
中断	输入	并行端口/中断	端口 B，引脚 0
模式选择开关，USART 接收器	输入	并行端口	端口 C，引脚 7
诊断 LED 灯 超频回声 USART 发送器	输出 输入 输出	并行端口	端口 C，引脚 6
诊断 LED 灯；超频脉冲	输出	并行端口	端口 C，引脚 5
I^2C，串行数据	双向	同步串行端口	端口 C，引脚 4
I^2C，串行时钟	双向	同步串行端口	端口 C，引脚 3
右电机 PWM 驱动；PWM 演示；TP 1 和 TP2	输出	PWM	端口 C，引脚 2
左电机 PWM 驱动	输出	PWM	端口 C，引脚 1
反射性光学传感器	输入	Timer 1	端口 C，引脚 0
左电机启用	输出	并行端口	端口 A，引脚 5
反射性光学传感器	输入	Timer 0	端口 A，引脚 4
光学传感器，后面	输入	模数转换器(ADC)	端口 A，引脚 3
右电机启用	输出	并行端口	端口 A，引脚 2
光学传感器，左边	输入	ADC	端口 A，引脚 1
光学传感器，右边	输入	ADC	端口 A，引脚 0

AGV 通用端口的分配具有一定的灵活性。只有施密特触发器输入、内部上拉电阻(端口 B)和微控制器的物理位置会影响到分配的选择。AGV 的开关和 LED 接口电路的实现在本节讨论,其余接口电路在后面讨论。

1. 开关接口

AGV 有一个模式选择开关,它是由用户来控制 AGV 在 2 种模式下转换;还有一对微开关,用于撞击传感。从电气上来说,这些开关都是 SPST(单刀单掷)型的,因此它们可以使用图 3-7 所示的方式进行连接。连接时,可以使用一个线性的上拉电阻或者直接利用端口 B 的上拉电阻。如果使用后面一种方法,在端口处于输入模式时,不管这些位是否需要上拉电阻,端口 B 所有位的上拉电阻都将被打开。如果外接电阻,只需要 3 个上拉电阻,因此不使用端口 B 自身的电阻。

2. LED 驱动

在电路中,有 2 个通用的诊断性 LED。程序员可以将它们应用于任何用途。由于在较低的电流下仍然能呈现出极佳的输出特性,我们选择高效能的红色 LED。试验表明,在 3.5mA 左右的电流下,LED 可到达一个合适的能见度。从图 3-8a 中的特性曲线可以得到,在此电流下,LED 的电压大约为 1.88V;从图 7-14a 中,可以观察到输出位的电压大约是 4.7V。因此,利用如下公式(3.1):

$$R=\frac{4.7-1.88}{0.0035}=806\Omega$$

820Ω 很接近这个“推荐值”,因此在电路中使用它来与 LED 串联。

7.10.3 硬件集成

如果你正在构建 Derbot AGV 项目,可以使用本书附属资源中的组件布局图在 AGV 中集成如下组件:

- ☐ 电源通路上的所有组件,包括去耦电容;
- ☐ 复位开关和上拉电阻;
- ☐ 晶体和负载电容;
- ☐ 诊断性 LED 以及当前条件下大小合适的电阻;
- ☐ 2 个微控制器开关以及上拉电阻;
- ☐ 模式选择开关和上拉电阻;
- ☐ (可选的)压电式发声器和驱动晶体管;
- ☐ 电路内调试器连接器(如果你有一个 ICD 2 单元)。

174

做完这些之后,你就可以得到一个 PCB,如图 7-22 所示。

175

图 7-22　AGV 中间集成阶段 1

7.11　使用 ICD 2 下载、测试和运行一个简单的程序

下面我们在新设计的硬件上使用 ICD 2(如果有的话)运行一个程序。

7.11.1　第一个 AGV 程序

例程 7-1 是用于测试前面构建的初始 AGV 的一个非常简单的程序。它的功能只是简单地测试 AGV 前面的微开关和诊断性 LED 的打开或者关闭功能。如果 2 个开关同时按下,发声器会发声。需要注意的是程序使用了 16F873A 的包含文件:

```
include p16f873a.inc
```

在较大的微控制器上进行编程时,输入寄存器的地址不再是一个实用的方法,如我们在例程 4-2 中开始处那样的做法。这种做法会在代码输入阶段引入错误,且错误

很难查出。使用包含文件应该成为你的习惯。首先，在 MPLAB 中创建一个 **Dbt_sw2led**（名字是任意的）项目。然后，从附属资源下载 **Dbt_sw2led.asm** 程序或者手工输入程序代码。程序的完整清单如图 7-1 所示。之后，使用通常的方式构建项目。如果你没有 ICD 2 电路内调试器，可以使用普通的编程器将程序下载到 PIC 控制器中，如第 4 章提到的 PICSTART Plus。

176

例程 7-1 AGV 上简单的移位操作

```
;***********************************************************
;Dbt_sw2led
;Moves state of front microswitches to diagnostic leds.
;If both switches on then buzzer goes on.
;TJW 24.3.05                    Tested & working 20.8.05
;***********************************************************
;
        list p=16f873a
        include p16f873a.inc

;Specify RAM labels
delcntr1 equ 20         ;used in delay SR
delcntr2 equ 21
;
;Set Configuration Word: crystal oscillator HS, WDT off,
;       power-up timer on, code protect off, LV Program off.

        __CONFIG _HS_OSC & _WDT_OFF & _PWRTE_ON & _LVP_OFF
        org 00
;Initialise
start   bsf   status,5      ;select memory bank 1
        movlw B'00000011'   ;all port A bits op
        movwf trisa
        movlw B'11111000'
        movwf trisb         ;port B bits
        movlw B'10000000'
        movwf trisc         ;port C bits
        movlw B'00000110'
        movwf adcon1        ;set port A for digital function
        bcf   status,5      ;select bank 0
;
;The "main" program starts here
;Switch all outputs off
        clrf  porta
        clrf  portb
        clrf  portc
;diagnostic, switch leds on for half a second
        bsf   portc,6
        bsf   portc,5
        call  delay500
        bcf   portc,6
        bcf   portc,5
        call  delay500
```

```
;move microswitch states to diag leds
loop bcf    portc,6        ;preclear port C, bit 6 (led off)
     btfsc portb,4         ;jump if switch pressed
     bsf    portc,6        ;led on if switch clear
;
     bcf    portc,5        ;preclear port C, bit 5 (led off)
     btfsc portb,5         ;jump if switch pressed
     bsf    portc,5        ;led on if switch clear
;
     btfsc portb,4
     goto   loop1
     btfsc portb,5
     goto   loop1
     bsf    portb,1        ;switch on sounder if both pressed
     goto   loop
loop1 bcf   portb,1        ;switch off sounder
     goto   loop
;*********************************************
;SUBROUTINES
;*********************************************
;introduces delay of 1ms approx
delay1  movlw D'250'       ;250 cycles called,
                           ;each taking 4us
        movwf delcntr1
del1    nop          ;4 inst cycles in this loop, ie 4us
        decfsz delcntr1,1
        goto del1
        return
;
;500ms delay (approx)       ;500 calls to delay1
delay500  movlw D'250'
       movwf  delcntr2
del5   call   delay1
       call   delay1
       decfsz delcntr2,1
       goto   del5
       return
       end
```

[177]

7.11.2 应用电路内调试器 ICD2

下面的介绍假设你有一个 ICD2,或者更新的版本。这是一个非常简要的介绍,这是由于假设你已经了解了 MPLAB 的特性,而 ICD 2 同样会使用这些特性。更进一步的详细介绍可以参阅完整的手册[7.3]。

请阅读手册的 Getting Started 小节,然后确保在你的主机和操作系统上正确配置 ICD 2。使用 ICD 2 将主机和 AGV 连接,如图 7-20 所示。打开 MPLAB 和你已经创建的包括例程 7-1 的项目。通过菜单 **Debugger**> **Select Tool** >**MPLAB ICD 2** 将 ICD 设置为调试模式。然后通过菜单 **Debugger** > **Connect** 完成主机和 AGV 之间的连接。ICD

会进行一个自测，检测电源是否存在以及能否将正确的电压应用到它所控制的电缆线。如果目标系统上电，并且正常运行，ICD会通过自测并识别出系统中的微控制器。在软件界面上，它会显示一条指示是否通过测试的信息，若通过测试，它会发出 **MPLAB ICD Ready** 消息。然后，**Debugger** 的下拉菜单变为有效，调试工具条也开始有效，如图 4-9 和图 7-23 所示。一个 ICD 2 额外的工具条也会出现，如图 7-23 的右边所示。它允许用户读程序存储器、下载程序和复位 ICD。但是需要注意，如果按下主工具条的 **Programmer** >**Select Programmer** 项，菜单会显示没有编程器被选择。这是由于在调试模式下，只能调用 ICD 中的选项对微控制器进行编程。

确保 AGV 已经打开，然后通过如图 7-23 中所示的工具条或者菜单 **Debugger**>**Programmer** 将程序下载到微控制器的程序存储器中。打开变量观察窗口(第 4 章已经讲述过)，它会显示寄存器 **PCL**、**PORTB** 和 **PORTC** 的值。然后对程序进行单步调试，使用 Step Over 命令跳过延时程序。一旦进入主循环程序段，试着依次按下微开关，然后在观察窗口中查看端口值的变化。这里与使用软件仿真器不同，我们看到的值是从硬件端口中传回的实际硬件值。你可以尝试在主机设置或者清除端口位，然后考虑一下你是如何能在主机上打开或者关闭 AGV 的 LED 灯的。与使用仿真器相同，可以双击程序的任意一行来设置断点。

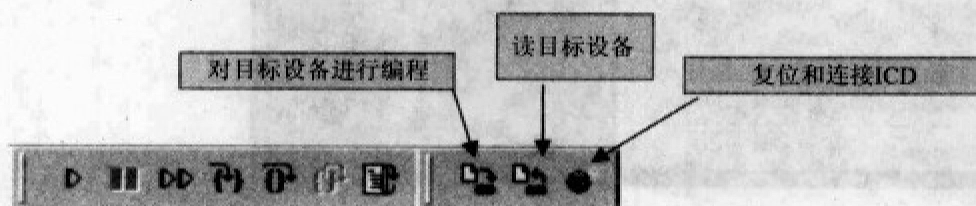


图 7-23 ICD2 的调试工具条——标准调试以及 ICD 额外的编程特性

为了对最终产品进行编程，ICD 2 必须转换到编程器模式。在工具条的顶部，单击 **Programmer** >**Select Programmer** > **MPLAB ICD 2**，然后单击 **Programmer** > **Connect**(一个自动连接模式也是可能的)。再次选择 Debugger 菜单，可以看到现在已经没有调试器可以选择了。在这个模式下，可以进行编程和读取程序存储器。可以使用相应的下拉菜单和编程工具条，如图 7-24 所示。注意，可以从 ICD 界面上控制复位。当编程过程完成，复位变为有效。可以在工具条上按下退出复位按钮，这样系统就退出复位，程序立即开始运行。

7.11.3 在程序中设置配置字

16F873A 比 16F84A 多了很多配置字位，如果在 MPLAB 软件窗口中设置它们是很容易出现错误的。错误的设置会导致系统出现异常的行为或者没有任何反应。但是，可以在程序中使用汇编指令_CONFIG 对配置字位进行设置。这条汇编指令在微控制器的包含文件中定义。针对 16F873A，相应的定义部分如例程 7-2 所示。每一个十

六进制的值都是从配置字的设置中获得(如图 7-7 所示)。将需要的配置方式对应的十六进制值相与,就可以获得最终想要的配置字。下面这段设置配置字的程序可以应用 179 于所有基于 16F873A 的 AGV 程序中。

```
;Set Configuration Word: crystal oscillator HS, WDT off,
;   power-up timer on, code protect off, LV Program off.
    __CONFIG _HS_OSC & _WDT_OFF & _PWRTE_ON & _LVP_OFF
```

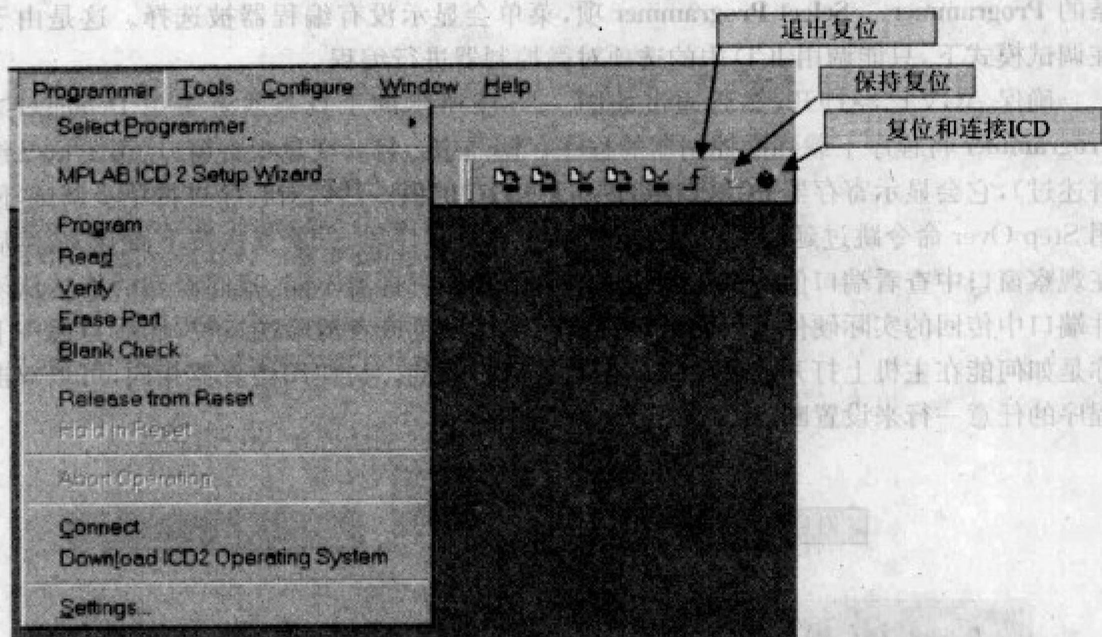


图 7-24 ICD 2 的编程器特性——下拉菜单和工具条

即使在程序中设置配置字,还是会出现一些错误。这种错误可能是在 MPLAB 配置字位窗口中对配置字不经意的修改。可以通过例程 7-1 来验证。可查看窗口中配置字位的设置,它们是正确的。但是,此时可能对窗口中配置字值进行修改,这会引入错误。在下一次程序下载时,窗口中的任何值都会被下载到程序存储器中。但是,我们可以在软件界面中取消 Configure > Settings > Programmer Loading 选项,并选择"Clear configuration bits upon loading the program"框。现在,当程序下载时,窗口中的配置字的设置将被清除,程序中的配置字位会被下载。

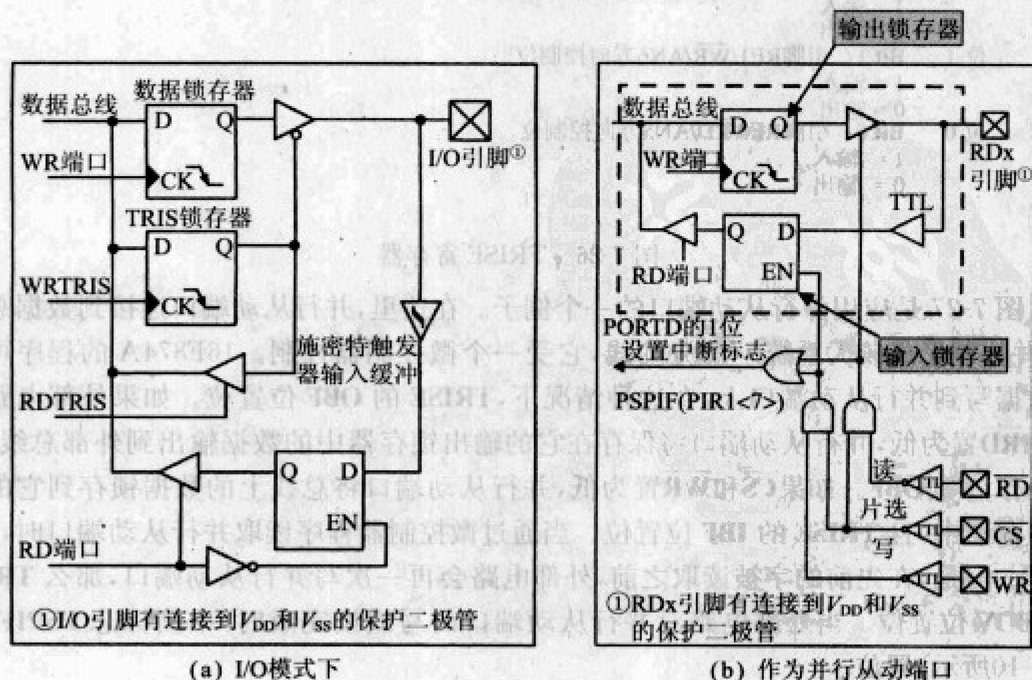
例程 7-2 在 16F873A 包含文件中定义配置字位

```
;==========================================================================
;
;     Configuration Bits
;
;==========================================================================
_CP_ALL                      EQU     H'1FFF'
_CP_OFF                      EQU     H'3FFF'
_DEBUG_OFF                   EQU     H'3FFF'
_DEBUG_ON                    EQU     H'37FF'
_WRT_OFF                     EQU     H'3FFF'  ;No prog memory write protection
```

```
_WRT_256        EQU     H'3DFF'  ;First 256 prog memory write protected
_WRT_1FOURTH    EQU     H'3BFF'  ;First 1/4 prog memory write protected
_WRT_HALF       EQU     H'39FF'  ;First half memory write protected
_CPD_OFF        EQU     H'3FFF'
_CPD_ON         EQU     H'3EFF'
_LVP_ON         EQU     H'3FFF'
_LVP_OFF        EQU     H'3F7F'
_BODEN_ON       EQU     H'3FFF'
_BODEN_OFF      EQU     H'3FBF'
_PWRTE_OFF      EQU     H'3FFF'
_PWRTE_ON       EQU     H'3FF7'
_WDT_ON         EQU     H'3FFF'
_WDT_OFF        EQU     H'3FFB'
_RC_OSC         EQU     H'3FFF'
_HS_OSC         EQU     H'3FFE'
_XT_OSC         EQU     H'3FFD'
_LP_OSC         EQU     H'3FFC'
```

7.12 深入了解 16F874A/16F877A 的端口 D 和端口 E

从图 7-1b 中可以看到 16F874A 和 16F877A 有 2 个额外的端口——8 位的端口 D 和 3 位的端口 E。与其他端口一样，每个端口都可用作通用 I/O。当配置成通常的数字 I/O 时，端口 D 的框图如图 7-25a 所示。端口 E 的另一个功能是提供 3 个模拟输入。因为这样，端口 E 也受控制 ADC 的其中一个寄存器（即 **ADCON1**）的控制。对该寄存器的设置决定了该端口是用于数字输入还是模拟输入。 180



(a) I/O模式下　　(b) 作为并行从动端口

图 7-25　端口 D 引脚驱动器电路框图（阴影框中所附标签为作者所加）

端口 D 和端口 E 合在一起也可形成并行从动端口。通过设置寄存器 **TRISE** 中的 **PSPMODE** 位(见图 7-26)使端口处于该模式下。该模式的作用是允许微控制器作为一个受其他微控制器控制的数据总线的从设备接口。端口 E 的各位必须设置为输入(在 **ADCON1** 中选择数字模式),但是 **TRISD** 的状态无关紧要。然后,将端口 D 和 E 配置成图 7-25b 中所示的样子。图中显示了端口 D的 1 位和 3 根控制线 $\overline{\text{CS}}$、$\overline{\text{WR}}$ 和 $\overline{\text{RD}}$。这 3 根控制线是按照并行从动目的配置为端口 E 的其中 3 位。注意端口 D 的每位都有一个输出锁存器和输入锁存器。

R-0	R-0	R/W-0	R/W-0	U-0	R/W-1	R/W-1	R/W-1
IBF	OBF	IBOV	PSPMODE	—	Bit 2	Bit 1	Bit 0

位 7 　　　　　　　　　　　　　　　　　　　　　　位 0

并行从动端口状态/控制位

位 7　**IBF**：输入缓冲满状态位
1 = 接收到一个数据，等待CPU读取
0 = 未接收到任何数据

位 6　**OBF**：输出缓冲满状态位
1 = 输出缓冲仍保存着上一次写入的数据
0 = 已读取输出缓冲

位 5　**IBOV**：输入缓冲溢出检测位(在微处理器模式下)
1 = 在上一次输入数据尚未读取前发生了一次写入(必须用软件清零)
0 = 无溢出发生

位 4　**PSPMODE**：并行从动端口模式选择位
1 = 并行从动端口模式中的PORTD功能
0 = 通用I/O口模式中的PORTD功能

位 3　未实现：读作“0”

PORTE 数据方向位

位 2　**Bit** 2：引脚RE2/$\overline{\text{CS}}$/AN7方向控制位
1 = 输入
0 = 输出

位 1　**Bit** 1：引脚RE1/$\overline{\text{WR}}$/AN6方向控制位
1 = 输入
0 = 输出

位 0　**Bit** 0：引脚RE0/$\overline{\text{RD}}$/AN5方向控制位
1 = 输入
0 = 输出

图 7-26　TRISE 寄存器

图 7-27 是应用并行从动端口的一个例子。在这里,并行从动端口连接到数据总线上,并控制形成较大系统一部分的线,它受一个微控制器控制。16F874A 的程序可以将数据写到并行从动端口上,在这种情况下,**TRISE** 的 **OBF** 位置位。如果外部电路将 $\overline{\text{CS}}$ 和 $\overline{\text{RD}}$ 置为低,并行从动端口将保存在它的输出锁存器中的数据输出到外部总线上,该动作清零 **OBF**。如果 $\overline{\text{CS}}$ 和 $\overline{\text{WR}}$ 置为低,并行从动端口将总线上的数据锁存到它的输入锁存器中,且 **TRISE** 的 **IBF** 位置位。当通过微控制器程序读取并行从动端口时,**IBF** 清零。但是,在先前的字被读取之前,外部电路会再一次写并行从动端口,那么 **TRISE** 的 **IBOV** 位置位。当外部电路对并行从动端口的写或读完成时,中断标志 **PSPIF**(如图 7-10所示)置位。

181

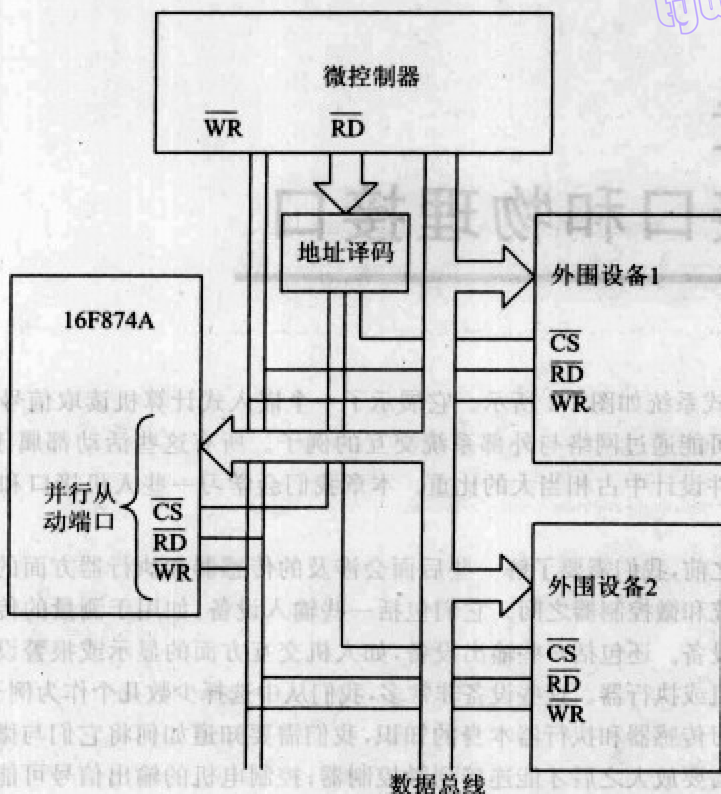


图 7-27 连接到系统总线的并行从动端口

小结

- ❑ 16F87XA 组是 16 系列型号微控制器的一个重要的子集。它增加了一组功能强大的外围设备且存储器得到了升级，因此功能更强。
- ❑ 16F87XA 组的核心体系结构与 PIC 16 系列的其他微控制器一样，采用的指令集也相同。 [182]
- ❑ 必须从系统地对嵌入式系统进行测试，而且要使用最好的工具来进行测试。
- ❑ 电路内调试是一种非常有用的测试和调试软硬件的技术，它对程序执行的干扰最小。

参考文献

7.1. PIC 16F87XA Data Sheet (2003). Microchip Technology Inc., DS39582B; www.microchip.com
7.2. PIC 16F87XA Flash Memory Programming Specification (2002). DS39589. www.microchip.com
7.3. MPLAB ICD 2 In-Circuit Debugger User's Guide (2005). Microchip Technology, DS51331B. [183]

第 8 章
人机接口和物理接口

典型的嵌入式系统如图 1-1 所示。它展示了一个嵌入式计算机读取信号、输出控制信号、同使用者交互和可能通过网络与外部系统交互的例子。所有这些活动都属于接口的功能，它在嵌入式系统硬件设计中占相当大的比重。本章我们会学习一些人机接口和物理接口方面的知识。

在设计接口之前，我们需要了解一些后面会涉及的传感器和执行器方面的知识。传感器和执行器位于大系统和微控制器之间。它们包括一些输入设备，如用于测量的传感器或用于人机交互的数据输入设备。还包括一些输出设备，如人机交互方面的显示或报警设备以及与物理系统交互方面的电机或执行器。这些设备非常多，我们从中选择少数几个作为例子来介绍。

为了深入学习传感器和执行器本身的知识，我们需要知道如何将它们与微控制器连接。传感器的信号可能需要放大之后才能连接到微控制器；控制电机的输出信号可能需要驱动功能强大的开关电路，以使电机能在正确的速度和时间下运行。

最后，学习如何编写程序来完成这些交互也是很重要的。

因此，在本章中你将学到：

- ☐ 人机交互的需求和实现交互的一些简单方法；
- ☐ 一些简单的传感器实例；
- ☐ 传感器信号与微控制器进行交互的一些方法；
- ☐ 一些简单的执行器实例；
- ☐ 执行器与微控制器进行交互的一些方法；
- ☐ Derbot AGV 中一些传感器和执行器的应用。

本章的许多主题都将借助 Derbot AGV 这个实例来讲述。

需要注意的是，接口的一些重要部分没有在本章出现。比如模拟信号的整个输入过程将在第 11 章讲述，由串行通信扩展的网络概念将在第 10 章介绍。

8.1 人机接口概述

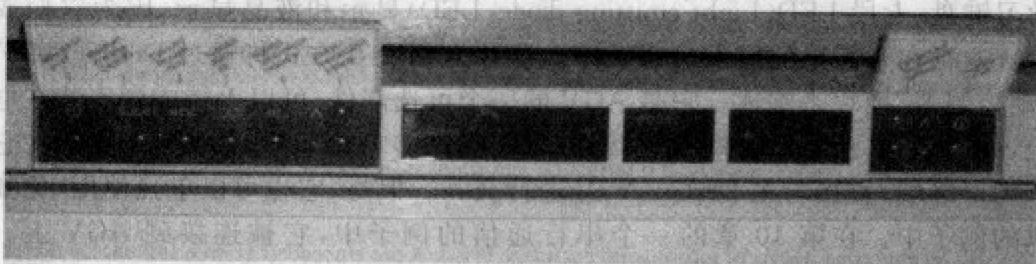
184

人必须与他使用的机器进行交互，这种交互几乎不可避免地具有闭环的特征。使用者感知机器的当前状态和可能需要了解的环境状态。通过这种行为，从系统中接收信息。然后，根据自己想要达到的目的，与机器进行交互，引发系统状态的改变。这种交互可能完全以信息交换的形式进行。例如，冰箱的使用者读取显示屏上显示的冰箱

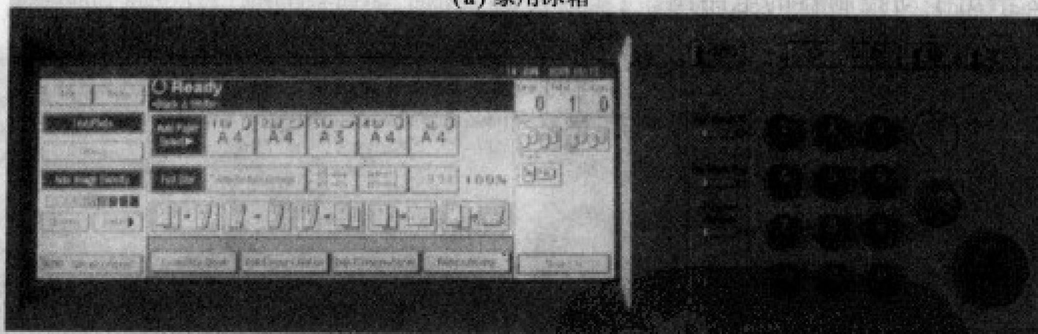
温度后觉得温度过高，于是通过键盘设定一个较低的温度。另外还有一种形式，使用者可能不得不对系统施加若干物理动作。这种情况经常出现在驾驶汽车时，驾驶员从仪表板上接收信息，但是之后仍然需要把自己的意图变成施加给汽车的动作比如转动方向盘、换档或者加速，来到达改变汽车状态的目的。

图 8-1 显示了一些家用电器中人机交互的例子，这些日常用品也都属于嵌入式系统。冰箱可工作于 2 种模式："超级"和"节能"。控制面板显示了冷冻室和冷藏室的实际温度。可以分别设置它们的温度。当温度超过一个特定的值时，声学报警器将会发声。这个报警器也可以被禁止。所有的控制信息都是通过一些简单的按钮进行输入，信息通过 2 个 2 位的数字指示器显示。

185



(a) 家用冰箱



(b) 复印机



(c) 汽车仪表板（局部）

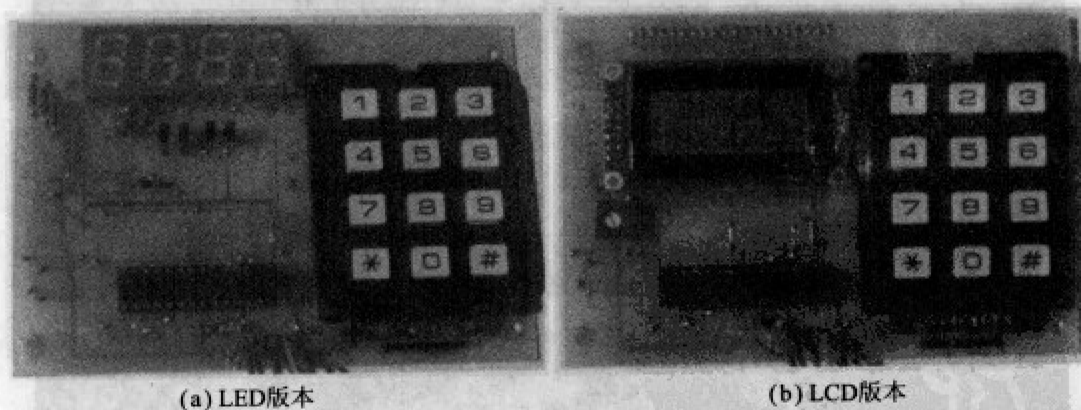
图 8-1 一些家用电器中的人机接口

复印机的控制稍微复杂一些。它可设置的操作模式比较多，如不同的纸张来源、图像调整功能等。可定制的显示面板用来显示信息。由于面板是触摸屏，因此它也可以用于控制信息的输入。传统的数字键盘用来输入纯粹的数字型数据，比如代码和复印的份数。

汽车仪表板的局部图是一个更复杂的例子。中间显示的是汽车的速度——由于在当前情况下，汽车是静止的，因此它显示出一个较大的 0。在它旁边，可以看到一系列的状态信息，包括引擎转速 r. p. m、电台、温度、车门状态和燃料情况。尽管信息复杂多样，它们都是以简单的形式显示出来的。这里再一次用到大量的数字显示器，如垒积的条形图、简单的发亮符号和图示信息。在图中可以发现没有输入部件。

在这 3 个特性各异的接口中，值得注意的是它们之间具有共同点。它们都具有显示数字化信息和状态信息的强大功能。实现这一点非常简单。例子中使用到的传感器主要是光学的。也使用了声学传感器，例如在报警时。

我们将在后续部分研究一些嵌入式系统和用户间进行信息交互的方法。首先会学习键盘、七段 LED(Light-emitting diode, LED) 显示和液晶显示，因为它们或多或少普遍存在于简单的嵌入式系统中。这些人机交互方式将通过 Derbot AGV 的"手动控制器"模块来阐述。它是 AGV 的一个可选部件，可以手持或者安装在电池盒上，如图 A3-3 所示。手动控制器有 LED 和 LCD 两个版本，如图 8-2 所示。它是一个无声终端，直接与主 Derbot AGV 进行交互。但是，它也可以单独使用，比如在本章的例子中。在第 10 章的一个串行通信的例子中，它被连接到 AGV 上。图 A3-2 给出手动控制器的电路图。



(a) LED版本　　(b) LCD版本

图 8-2　Derbot AGV 中的手动控制器

8.2　从开关到键盘

第 3 章中已经介绍了简单的开关，它们是许多人机接口的主要组成部分。开关适合于用来传递数字信息——它们有两个状态，也可以使用多个开关来表示多状态，此时一个开关代表一个端口位。但是当设计复杂时，不适合持续地增加开关的数量来满足状态的需求。首先，如果端口位数需求非常多时，开关的这种严格的二级制特征通常是不能满足这种需求的。

8.2.1 键盘

键盘是开关的一个有用扩展，如复印机接口和图8-2中的AGV手动控制器。键盘允许输入数字或者字母。它广泛地使用于复印机、防盗警报器和中央暖气系统控制器等中。

键盘是基于开关的，但是如果键盘的每一个开关都分配一个端口值，键盘将会极度耗费资源。为了更好地利用资源，开关被连接成矩阵形式。图8-3就是图8-2中键盘的电路开关连接图，它有12个按键。开关被排列成一个4行3列的4×3矩阵。现在只需要7个端口位，而不是12个。任何一个按键被按下，将接通它所在的行和列。

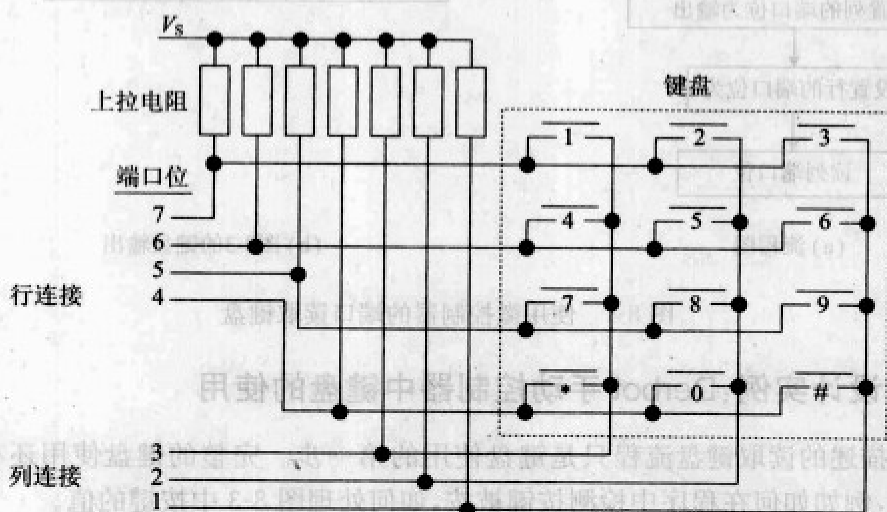


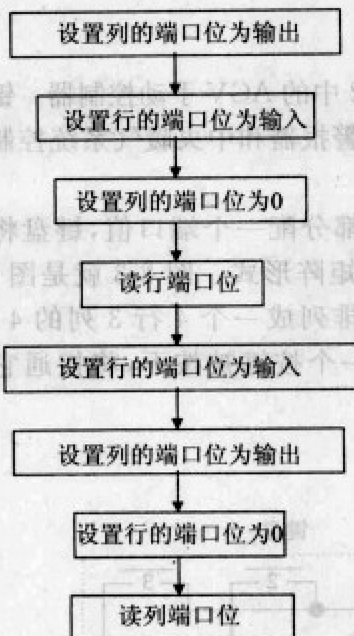
图8-3 键盘电路(含有上拉电阻)

187

在嵌入式系统中，键盘通常被连接到微控制器的输入/输出(I/O)端口上。它们的连接如图所示，其中还有一些必要的上拉电阻。如何检测哪个按键被按下是一个挑战。

读取键盘的通用方法如图8-4a中的流程图所示。首先，行的端口位设置为输出模式，列的端口位设置为输入。行的输出端口位设置为0。如果没有按键被按下，由于上拉电阻的作用，所有的行线输入值为1。但是，如果某一个按键被按下，那么它相应的开关将接通所在的行线和列线，相应的行线将被拉低。立即重复相同的过程，但是将行设置为输入，列设置为输出，就可以检测到按键所在的列线，从而识别出被按下的按键。

图8-3中键盘的端口值如图8-4b所示。例如，假设7键被按下。在流程图的第1个阶段，行作为输入，第3行线(端口的位5)将读为低。第2个阶段中，列作为输入，第1列线(端口的位3)将读为低。最终，7键的端口值如图所示，其中位0是无关值，没有使用它。



(a) 流程图

按键	键盘端口值
1	0111 011×
2	0111 101×
3	0111 110×
4	1011 011×
5	1011 101×
6	1011 110×
7	1101 011×
8	1101 101×
9	1110 110×
*	1110 011×
0	1110 101×
#	1110 110×

(b) 图8-3的键盘输出

图 8-4　使用微控制器的端口读取键盘

8.2.2　设计实例：Derbot 手动控制器中键盘的使用

上面描述的读取键盘流程只是键盘使用的第一步。完整的键盘使用还有许多其他的步骤，例如如何在程序中检测按键被按，如何处理图 8-3 中按键的值。

188

例程 8-1（程序源文件名为 keypad_test）是上一小节键盘读取的实际应用。我们把它作为例子来阐述键盘和 LCD 的使用。程序代码很多，这里只列出与键盘相关的部分。本书的附属资源中包含完整的代码。

图 8-5 为程序的流程图。为了检测键盘的动作，程序使用了端口 B 高 4 位的“电平变化中断”功能。初始化之后，程序的所有活动都包含在名为 **kpad_to_lcd** 的中断服务程序（Interrupt Service Routine，ISR）中。它本质上做了 4 件事情：读取键盘端口值、转换端口值为 ASCII 码、输出 ASCII 码到 LCD，以及在离开 ISR 之前等待按键被释放。

可以试着完整地跟踪一下程序，然后与图示的程序流程进行比较。在程序开始的初始化部分，端口 B 的行线设置成输入，列线为输出。我们本来也可将行线设置为输出，列线设置为输入，但是为了利用“电平变化中断”功能，必须设置端口 B 的高 4 位为输入。接下来，设置所有的输出端口位为 0，并且启用“电平变化中断”。在这个阶段我们实际上已经进入了图 8-4a 的流程图中。

当按键被按下时，中断被触发。中断服务程序 **kpad_to_lcd** 被调用。在程序开始处，首先调用子例程 **kpad_rd**，它将继续执行由程序初始化时启动的图 8-4a 中键盘读取

的流程。端口 B 的值被读取，并存放在内存单元 **kpad_pat** 中。然后调换行列的输入输出模式，之后读取行端口的值。行值与 0 相与是为了清零无关位，之后与内存单元 **kpad_pat** 的值相或再放回原来的位置。程序然后将端口 B 复位为初始值，准备下一次读取键盘。

189

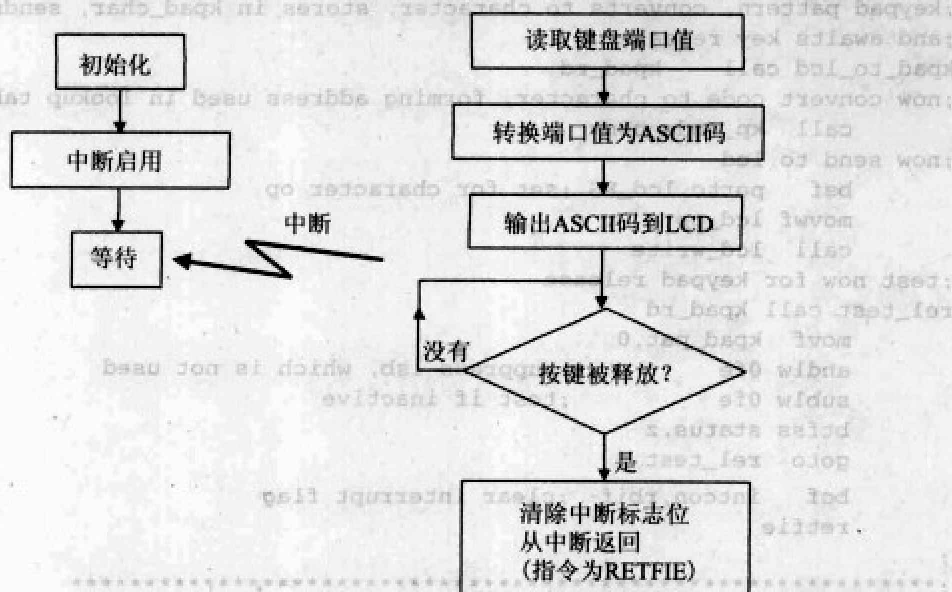


图 8-5　例程 8-1 的程序流程图

例程 8-1　Derbot 手动控制器中的键盘读取

```
;***************************************************************
;keypad_test
;Tests keypad, writing key pressed to lcd display on
;Derbot Hand Controller.
;TJW 23.6.05                                    Tested 24.6.05
;***************************************************************
...
(opening program sections omitted)
...
;Initialise
        bsf    status,rp0      ;select memory bank 1
...
        movlw B'11110000'      ;Port B initially Row bits ip, column op
        movwf trisb            ;(port B not used)
        bcf    status,rp0      ;select bank 0
...
(lcd initialisation omitted)
...
        clrf   portb ;initialise keypad value
;enable interrupt
        bcf    intcon,rbif
        bsf    intcon,rbie
        bsf    intcon,gie
loop    goto   loop            ;await keypad entries
```

```
;***********************************************************************
;Interrupt Service Routine.
;***********************************************************************
;Keypad press has been detected through Port B Interrupt on Change.Gets
;keypad pattern, converts to character, stores in kpad_char, sends to lcd,
;and awaits key release.
kpad_to_lcd call    kpad_rd
;now convert code to character, forming address used in lookup table
        call  kp_code_conv
;now send to lcd
        bsf   portc,lcd_RS ;set for character op
        movwf lcd_op
        call  lcd_write
;test now for keypad release
rel_test call kpad_rd
        movf  kpad_pat,0
        andlw 0fe           ;suppress lsb, which is not used
        sublw 0fe           ;test if inactive
        btfss status,z
        goto  rel_test
        bcf   intcon,rbif  ;clear interrupt flag
        retfie
;
;*************************************************************
;SUBROUTINES
;*************************************************************
;Reads keypad, places pattern into kpad_pat, and resets keypad interface
kpad_rd movf portb,w      ;read portb value, this will be row pattern
        andlw B'11110000'   ;ensure unwanted bits are suppressed
        movwf kpad_pat
        bsf   status,rp0    ;set row to op, column to ip
        movlw B'00001110'
        movwf trisb
        bcf   status,rp0
        movlw 00
        movwf portb         ;ensure output values still zero
        movf  portb,w       ;read portb value, this will be column pattern
        andlw B'00001110'   ;ensure unwanted bits are suppressed
        iorwf kpad_pat,1    ;OR those results into the pattern
;reset keypad interface
        bsf   status,rp0 ;set row to ip, column to op
        movlw B'11110000'
        movwf trisb
        bcf   status,rp0
        clrf  portb  ;ensure output values still zero
        return

;Converts keypad pattern held in kpad_pat to ASCII character, first forming
;address (in kpad_add) that is used in lu table. Returns with character held
;in kpad_char
kp_code_conv bcf status,c
```

```
        rrf    kpad_pat,1      ;discard bit 0 which is not used
        clrf   kpad_add
;deduce row
        btfsc kpad_pat,6
        goto   kp1
        goto   col_find        ;here if row 1, kpad_add stays as is
kp1     btfsc kpad_pat,5
        goto   kp2
        movlw B'00000100'      ;here if row 2
        iorwf kpad_add,1       ;form table address
        goto   col_find
kp2     btfsc kpad_pat,4
        goto   kp3
        movlw B'00001000'      ;here if row 3
        iorwf kpad_add,1       ;form table address
        goto   col_find
kp3     btfsc kpad_pat,3
        goto   kp4
        movlw B'00001100'      ;here if row 3
        iorwf kpad_add,1       ;form table address
        goto   col_find
kp4     movlw D'16'            ;no row detected, return "E" via Table
        goto keypad_op
;now deduce column
col_find btfsc        kpad_pat,2
        goto   cf1
        goto    keypad_op      ;here if column 1, kpad_add stays as is
cf1     btfsc kpad_pat,1
        goto   cf2
        movlw B'00000001'      ;here if column 2
        iorwf kpad_add,1       ;form table address
        goto   keypad_op
;assume now column 3
cf2     movlw B'00000010'
        iorwf kpad_add,1       ;form table address
keypad_op movf kpad_add,0
        call   kp_table
        movwf kpad_char        ;save the character
        return
;
;Table called to convert pattern recd from keypad to actual character. Note that
;ASCII codes will be returned, as each digit is in format 'D'.
kp_table addwf pcl,1
        retlw '1'              ;row 1
        retlw '2'
        retlw '3'
        retlw 'A'              ;Error code
        retlw '4'              ;row 2
        retlw '5'
        retlw '6'
        retlw 'B'              ;Error code
        retlw '7'              ;row 3
        retlw '8'
        retlw '9'
```

191

```
        retlw 'C'          ;Error code
        retlw '*'          ;row 4
        retlw '0'
        retlw '#'
        retlw 'D'          ;Error code
        retlw 'E'          ;Error code
...
```

执行完程序 **kpad_rd** 后，内存单元 **kpad_pat** 中的值还不是一个有用的格式。它是一个 7 位的数，可能的取值如图 8-4b 所示。因此，中断服务程序接着调用子例程 **kp_code_conv**，将这个数转换为产生它的键盘码。下面一段将描述这种转换所采用的方法。然后程序调用子例程 **lcd_write** 将键盘码输出到 LCD 进行显示。程序在这里没有列出这个过程，但是在本章后续部分会讲到。接着程序进入循环等待按键被释放。它再次调用子例程 **kpad_rd** 来查询按键状态。由于用户释放按键也会引起"电平变化中断"，如果不进入一段循环来检测按键是否释放就立即中断返回，这将会导致第 2 次不想要的键盘读取。

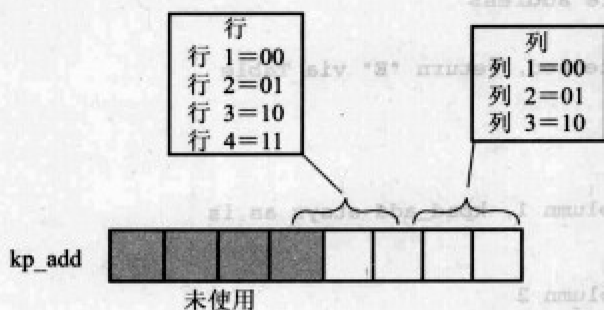


图 8-6 查找表的索引格式

kp_code_conv 子例程将从键盘读取的端口值(如图 8-4b 所示)转换为按键的 ASCII 码。它首先计算出存放在内存单元 **kpad_add** 中的一个变量，它是用来访问查找表 **kp_table** 的索引，格式如图 8-6 所示。子例程依次测试键盘端口值，找出键盘的哪一行哪一列被按下，然后根据测试的结果来设置访问查找表的索引。随后，调用查找表，如第 5 章所述。

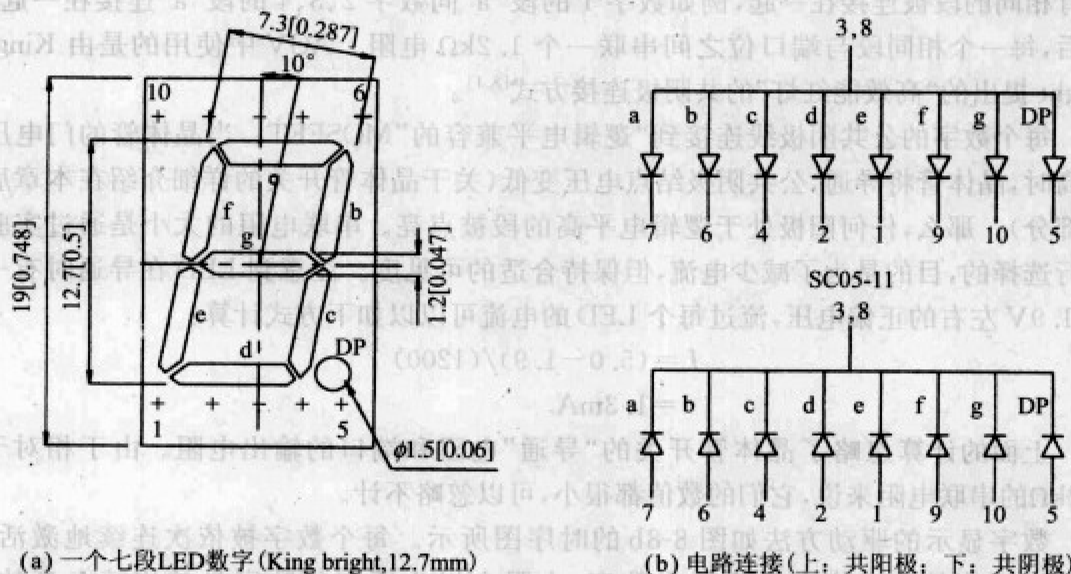
8.3 LED 显示

本章第一部分的几个例子用图形的方式说明了在几乎任何具有人机交互的系统中，显示功能是非常重要的。下面我们就来学习 2 种显示方式：七段 LED 显示和液晶显示。

8.3.1 LED 阵列：七段 LED 显示

简单的 LED 与开关一起也在第 3 章进行了介绍。我们可以发现 LED 的一些优点：在视觉上很吸引人，是一种高效能光源，可以由逻辑门或者端口位的输出来驱动，非常适合用来传递简单的信息。但是单个 LED，或者即使是一组 LED，能够传递的信息量受限。并且随着 LED 数量的增加，驱动电路将变得复杂。因此，人们设计了很多标准的 LED 结构，包括条形图、七段 LED 显示、点阵以及"星射状"。

七段 LED 显示方式是一种特殊的万能结构，如图 8-1a、图 8-1b 和图 8-2a 所示。本节我们将更深入地研究一下 LED 显示的实现。由 Kingbright[8.1] 设计的单个数字 LED 显示结构如图 8-7 所示。Derbot 手动控制器采纳了这种数字显示方法（如图 8-2a 所示）。通过点亮七段 LED 的不同组合，可以显示所有的数字和一些字符。单个数字通常还要包括一个小数点 LED，如图所示。难题出现了：如果每一段由 1 个 LED 点亮，1 个数字就需要 14 个电路连接。如果有多个数字，那么连接的数目就会剧增。因此，下面介绍 2 个巧妙而简单的技术用来减少连接的数目。



(a) 一个七段LED数字(King bright,12.7mm)　　(b) 电路连接(上：共阳极；下：共阴极)

图 8-7　七段 LED 显示(复制权得到 Kingbright Elec. 公司惠允)

1. 共阳极/共阴极连接

当以简单的形式使用多个 LED 时，几乎可以确定的是所有 LED 接线端的一个“边”都要与所有其他 LED 同类型的边连接。即所有 LED 的阳极或者阴极连接在一起。七段 LED 数字显示就使用了这种做法，如图 8-7b 所示。数字可以采用共阳极或者共阴极形式。一个数字包括 8 个 LED(包括小数点)。采用共阳极或者共阴极连接方式时，每一个 LED 占一个连接，一个公共极占一个连接，这时不再需要 16 个连接，而仅仅需要 9 个连接就可实现。例子中的实际引脚连接分布在数字的顶端和底端 2 排。因此，一个公共的阳极或阴极需要 2 个引脚，一个数字总计有 10 个连接。

2. 数字复用

以共阳极或者共阴极的形式进行连接时，单个数字的连接数目会大大减少。但是很少会使用单个数字，而多个数字仍然需要数量很大的连接。例如，一个 4 位的数字显示(每个数字都有一个小数点)将需要 36 个连接。此外，多个数字显示对电源要求也非常高。当所有段被点亮时，如果流过一个 LED 的电流为 5mA，公共极的电流将上升至 160mA。因此，我们引入第 2 个重要的技术——数字复用。这个技术使我们能够

开发出一个实际的数字显示，同时它也提出了在处理实时问题方面的一个有趣的前期挑战。通过下面的例子我们来讲述数字复用技术。

8.3.2 设计实例：在 Derbot 手动控制器中使用七段 LED 显示

由于 Derbot 手动控制器可支持 LED 显示或者 LCD(但是不能同时支持这 2 种)，Derbot 手动控制器的电路图(如图 A3-2 所示)相当复杂。因此，图 8-8a 只包含七段 LED 显示的一部分，我们从这个 LED 显示中来理解数字复用技术。从图中可以发现所有相同的段被连接在一起，例如数字 1 的段"a"同数字 2、3、4 的段"a"连接在一起。然后，每一个相同段与端口位之间串联一个 1.2kΩ 电阻。AGV 中使用的是由 Kingbright 提出的"高效能红灯"的共阴极连接方式[8.1]。

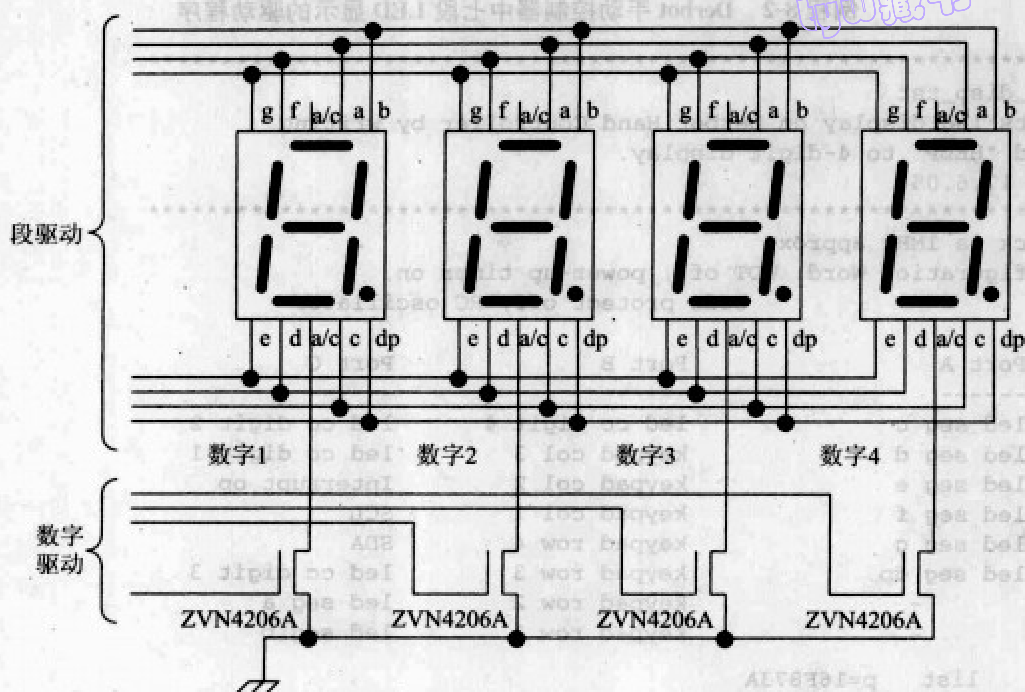
每个数字的公共阴极线连接到"逻辑电平兼容的"MOSFET。当晶体管的门电压变高时，晶体管将导通，公共阴极结点电压变低(关于晶体管开关的详细介绍在本章后续部分)。那么，任何阳极处于逻辑电平高的段被点亮。串联电阻的大小是通过实验进行选择的，目的是为了减少电流，但保持合适的可见度。注意到 LED 在导通时有一个 1.9V 左右的正偏电压，流过每个 LED 的电流可以以如下方式计算：

$$I=(5.0-1.9)/(1200)$$
$$=1.3\text{mA}$$

上面的计算忽略了晶体管开关的"导通"电阻和端口的输出电阻。由于相对于 1.2kΩ的串联电阻来说，它们的数值都很小，可以忽略不计。

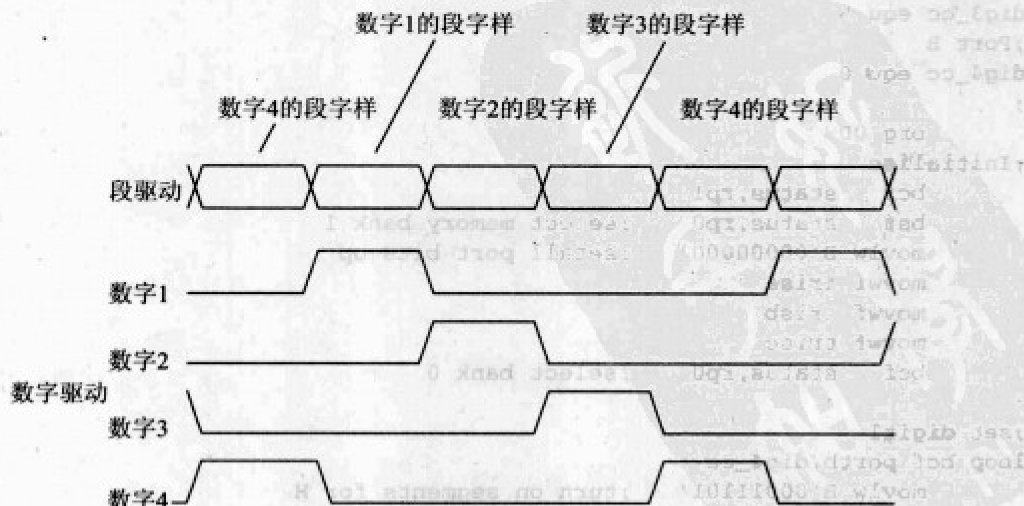
数字显示的驱动方法如图 8-8b 的时序图所示。每个数字被依次连续地激活。如果以适当的速度依次激活每个数字，人眼会以为所有的数字是被连续点亮的。在时序图中，数字 4 的段首先被设置，数字 4 的公共阴极被设置为 1。因此，数字 4 的显示被打开，其他数字的显示被关闭。这会持续一段时间(5ms～20ms 左右是一个合适的值)，然后数字 2 以相同的方式被点亮。每个数字在一个显示循环期间轮流被点亮。

如果由微控制器来驱动，需要编写一个程序来产生这个驱动时序。我们来看一下在例程 8-2 中实际上是如何产生这个时序的。它是运行在 LED 版本的 Derbot 手动控制器上的一个简单的程序，运行结果是在 4 个数字上显示字符 HELP。如图 8-7a 所示，对于字符 H，段 b、段 c、段 e、段 f 被点亮。简单的初始化之后，字符 H 的端口位字样被送到端口 A 和端口 C。在端口位字样中，与"点亮"的段连接的端口位被置为高。数字 1 的共阴极也被置高，然后调用一个延时 5ms 的程序。随后数字 2 的段字样(将显示字符 E)和公共阴极驱动取代了数字 1 的段字样和公共阴极驱动。程序在循环到重新开始之前，将继续运行直到显示完所有的 4 个字符。



段	端口位	段	端口位	数字驱动	端口位
a	C,6	e	A,2	1	C,0
b	C,7	f	A,3	2	C,1
c	A,0	g	A,4	3	C,5
d	A,1	d.p.	A,5	4	B,0

(a) 显示电路图细节



(b) 驱动数字的时序图

图 8-8 Derbot 手动控制器的七段 LED 显示部分

例程 8-2 Derbot 手动控制器中七段 LED 显示的驱动程序


```
;*************************************************************
;led_disp_tst
;Tests led display on Derbot Hand Controller by writing
;word "HELP" to 4-digit display.
;TJW 17.6.05
;*************************************************************
;Clock is 1MHz approx
;Configuration Word: WDT off, power-up timer on,
;                     code protect off, RC oscillator
;
;    Port A                  Port B              Port C
;    ------                  ------              ------
;0   led seg c               led cc digit 4      led cc digit 2
;1   led seg d               keypad col 3        led cc digit 1
;2   led seg e               keypad col 2        Interrupt op
;3   led seg f               keypad col 1        SCL
;4   led seg g               keypad row 4        SDA
;5   led seg dp              keypad row 3        led cc digit 3
;6      -                    keypad row 2        led seg a
;7      -                    keypad row 1        led seg b

        list    p=16F873A
        #include p16f873A.inc

;Specify RAM
delcntr1 equ 20         ;used in delay5
;Specify some port bits
;Port C

dig1_cc equ 1           ;digit common cathode drives
dig2_cc equ 0
dig3_cc equ 5
;Port B
dig4_cc equ 0
;
        org 00
;Initialise
        bcf   status,rp1
        bsf   status,rp0     ;select memory bank 1
        movlw B'00000000'    ;setall port bits op
        movwf trisa
        movwf trisb
        movwf trisc
        bcf   status,rp0     ;select bank 0
;
;set digit1
loop bcf portb,dig4_cc
        movlw B'00011101'    ;turn on segments for H
        movwf porta
        bcf   portc,6
        bsf   portc,7
        bsf   portc,dig1_cc    ;enable digit once segments set
        call delay5
```

195~196

```
;digit2
        bcf   portc,dig1_cc
        movlw B'00011110'    ;turn on segments for E
        movwf porta
        bsf   portc,6
        bcf   portc,7
        bsf   portc,dig2_cc ;enable digit
        call delay5
;digit3
        bcf   portc,dig2_cc
        movlw B'00001110'    ;turn on segments for L
        movwf porta
        bcf   portc,6
        bcf   portc,7
        bsf   portc,dig3_cc  ;enable digit
        call delay5
;digit4
        bcf portc,dig3_cc
        movlw B'00011100'    ;turn on segments for P
        movwf porta
        bsf   portc,6
        bsf   portc,7
        bsf   portb,dig4_cc ;enable digit
        call delay5
        goto loop
;
;SUBROUTINE: Introduces delay of 5ms approx
delay5 movlw D'250' ;250 cycles called
       movwf  delcntr1
dell   nop                     ;5 inst cycles in this loop, ie 2ous
        nop
       decfsz delcntr1,1
       goto   dell
       return
        end
```

197

图 8-9 是当程序运行时由 Agilent 混合信号示波器的逻辑分析仪部分截获的屏幕截图。图中显示出所有段的连接线以及 4 个数字的驱动波形。当一个数字的共阴极驱动为高时，对应的共阴极连接本身电压变低，数字将被激活。对于数字 1，很容易发现段 b、段 c、段 e、段 f、段 g 被点亮形成字符“H”。类似地，对于数字 2，段 a、段 d、段 e、段 f、段 g 被点亮形成字符 E。这个过程重复进行，最后拼写出单词 HELP。从图中可以看到段 e 和段 f 被连续点亮，但是表示小数点的 LED 总是关闭的。

198

LED 非常有用，但是也有一些缺点：至少对于电池供电的设计来说，它们非常耗电，所以很难（如果不是不可能的话）利用它们来进行复杂的多数字或者图形的显示。因此，寻找更先进的显示技术非常重要，下一章我们将介绍 LCD 技术。

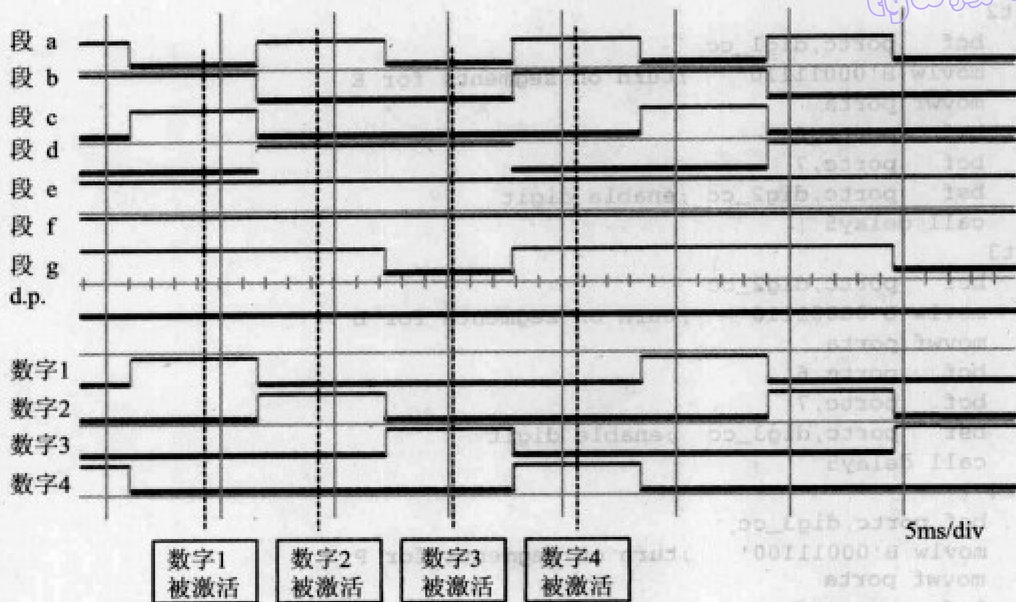


图 8-9 七段 LED 显示的输出波形——显示 HELP

8.4 LCD

LCD 是当前电子产品更新换代的一个可行技术之一。它是移动电话、笔记本和个人数字助理的必要组成部分。

液晶是一个有机化合物,它能够极化穿过它的任何光。当附加一个电场时,液晶会通过改变其分子的排列,随之改变穿过它的光被极化的方向。液晶可以被封装于两块平行的玻璃平板之间,平板上要带有图样匹配的透明电极。当在电极上加一个电压,晶体的光学特性改变,电极的图样会出现在晶体中。

目前,人们已经开发了很多种 LCD 方式,包括七段数字、点阵和各种各样的图形显示。许多通用的 LCD 显示器可以通过商业渠道获得,而一些定制的显示器用于大批量的产品中。在 Derbot 手动控制器中,LCD 显示器就是一个非常流行并且通用的 LCD 形式,如图 8-2b 所示。图中的显示器有两行,每行有 8 个数字,每个数字都是一个液晶点阵。其他这种形式的 LCD 显示器在结构上也是一样,只是包含更多的行和数字。

直接驱动 LCD 显示器并不是很简单。但是在这里我们并不用担心,因为大部分的显示器本身包含一个驱动电路用来与微控制器进行交互,例如图 8-2b 中的 LCD 显示器。因此,我们只需要知道如何与驱动电路进行交互就可以了。

8.4.1 HD44780 驱动和它的衍生电路

前些年,电子界巨人日立公司特别设计了一款微控制器用来驱动 LCD 的文字数字模块,如图 8-2b 所示的 LCD。它有一个简单的接口用来与通用微处理器或者微控制

器进行交互。HD44780[8.2]这款微控制器定义了一个接口，以后它几乎成为这种类型显示器的一个非成文的接口标准。许多显示器的制造商将这个接口集成到它们自己的产品中。现在它的一些衍生电路取代了日立原来的接口，但是仍然保留了原来大部分的特性。这些衍生电路包括三星公司的 S6A009 和 KS0066U 器件。由于这些衍生电路的特性与最初日立的设计很相似，因此本节将描述 HD44780 的特性。当我们的设计与 LCD 相关时，确保获得正确的器件数据，这是非常重要的。

在一定程度上，由于显示器本身的时间刻度约束，HD44780 接口具有一些奇特的性质。它具有以下一些重要的特征。

- 数据在 4 位或者 8 位数据总线上传输，由用户来配置传输位宽。数据可能是指令或者字符信息。当配置为 4 位传输模式时，整个接口电路只需要 7 根线就可以正常工作，但是数据传输的速度会比较慢。数据总线的第 7 位被复用为"忙标志"，指示器件是否可以接收新的数据。这个标志位很重要，因为 HD44780 大部分的操作都具有固定的完成时间，一个操作完成之后才能接受下一条指令。

199

- 控制由 3 根线来完成：
  - 寄存器选择(Register Select，RS)，确定传输的是指令还是数据；
  - 读/$\overline{写}$(R/$\overline{W}$)，确定数据方向；
  - 启用(E)，提供时钟来同步数据传输。
- 具有一个简单的指令集，用来对操作方式进行控制。包括初始化显示器、清屏、控制光标的形状和位置。
- 用户可以存取 2 个寄存器，由 RS 的值来确定存取的是哪个寄存器，这 2 个寄存器是：
  - 指令寄存器，用来传输指令(当 RS=0 时)；
  - 数据寄存器，用来传输要显示的数据，如字符码(当 RS=1 时)。
- 内部资源包括一个 80 个字节的显存(display RAM)和一个字符产生 ROM。

系统上电后，HD44780 必须执行一个很特殊的初始化过程。确保这段初始化过程完全正确是非常重要的，否则任何 HD44780 驱动的显示器都不能正常工作。任何一款基于这种接口的商业显示器将会在数据手册中给出一段用于初始化的指令序列。参考文献 8.3 给出了一个例子。初始化指令独立于显示器的显示位宽(4 位或者 8 位)，而且初始化时设备的忙标志位也不能使用。因此，这些初始化的指令通常被一个具有适当延迟时间的延时程序隔开，以保证一个操作完成之后才启动下一个操作。

由于受限于 LCD 本身的时序要求，HD44780 的操作速度要比大多数微控制器慢。因此，与它交互必须要处理一些特殊的时序问题。图 8-10 为一个配置为 8 位模式时接口的时序图。E 线上的每次脉冲表示一次到 LCD 控制器的数据传输。开始时 RS 设置为低，微控制器数据总线上的数据被解释为指令，HD44870 接收并执行它们。微控

制器需要了解指令是否被执行完毕。因此，设置 R/$\overline{W}$ 线为高，在 E 线的下一个脉冲时，LCD 控制器将输出一个字到数据总线。这个字的最高有效位为忙标志，低位为内部显存 RAM 的地址计数器。在忙标志被清除之前，不能再传输数据到 LCD 控制器。在这个例子中，当忙标志为低时，RS 变高，R/$\overline{W}$ 变低，因此下一次传输的数据将是一个字符码。

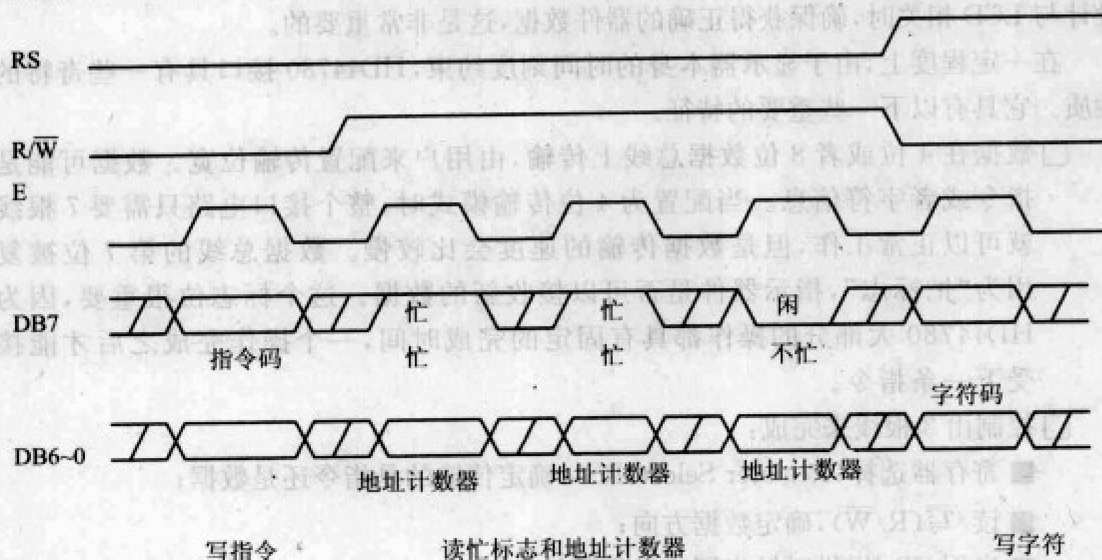


图 8-10 HD44780 设置为 8 位接口模式时的时序图

8.4.2 设计实例：在 Derbot 手动控制器中使用 LCD 显示器

LCD 版本的 Derbot 手动控制器中使用的是 Powertip 公司的 PC0802-A 型号的 LCD 显示器。它有 2 行，每一行有 8 个字符。S6A0069 作为驱动微控制器来控制这个显示器，它具有与 HD44780 相同的特性，使用 8 位的接口模式。图 A3-2 给出了整个芯片的电路图，图 8-11 只包含 LCD 部分。

200

用来演示 LCD 显示器的程序为 **keypad_test**，在例程 8-1 中我们已经引用这个程序。在程序中，当一个按键被按下时，程序会识别出按键，并且将这个按键的代码（ASCII 码）传输到 LCD 显示器。子例程 **lcd_write** 用来写一个指令或者字符码到显示控制器；子例程 **busy_check** 用来检查忙标志，这 2 个程序的代码在例程 8-3 中。可以核对本书附属资源中完整的程序列表清单，找到显示器初始化代码。

子例程 **lcd_write** 以调用 **busy_check** 开始，因此，显示控制器在准备好接收数据之前，不会发生对它的写操作。然后，程序设置 R/$\overline{W}$ 线为低，这指示一个写过程将要发生。由于在硬件中把连接到 LCD 控制器的 8 位数据线分配到 2 个端口（见图 8-11），程序中需要执行 2 个右移操作，导致程序有一点复杂。一旦数据在端口准备好，将使能线（程序中名称为 **lcd_E**）置高一个脉冲来完成传输过程。

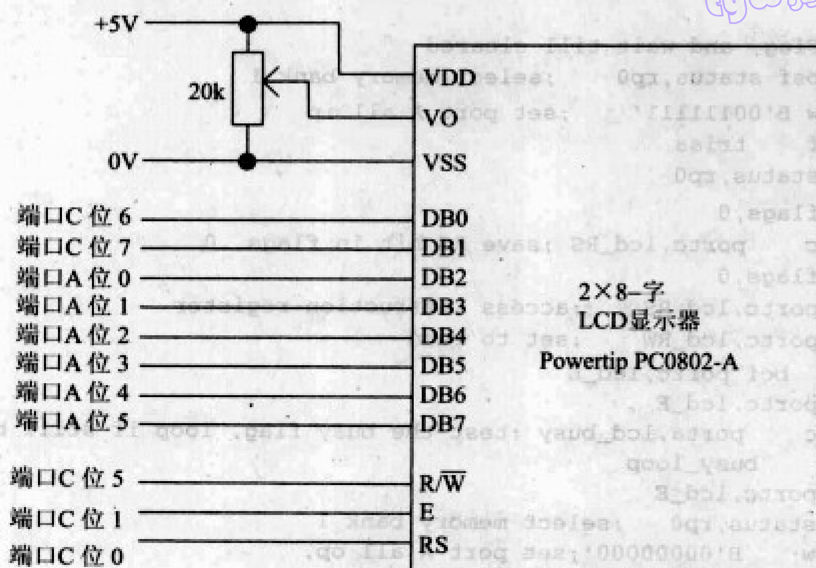


图 8-11 Powertip PC0802 LCD 显示器的连接图，连接到 Derbot 手动控制器

busy_check 子例程首先设置端口 A 为输入模式。然后程序保存 RS 的值，这是由于 RS 可以处于任何逻辑状态。随后 RS 被置低，R/$\overline{W}$ 被置高，这些操作是读忙标志需要设置的条件。E 线被周期性置高，并对忙标志进行一次测试。子例程一直循环直到忙标志被清除。

例程 8-3 键盘测试程序：LCD 驱动子例程

```
;Waits until busy clear, and writes word held in lcd_op to display.
;RS must be preset to required value, this status is preserved.
lcd_write call busy_check
        bcf portc,lcd_rw
        bcf status,c
        rrf lcd_op,1     ;form output bits, op word sits
;across ports a & c
        bcf portc,6 ;set value of bit 0 of bus
        btfsc    status,c
        bsf portc,6
        bcf status,c
        rrf lcd_op,1
        bcf portc,7 ;set value of bit 1 of bus
        btfsc    status,c
        bsf portc,7
        movf     lcd_op,0
        movwf    porta
        bsf portc,lcd_E
        bcf portc,lcd_E
        return
```

```
;
;Test Busy Flag, and wait till cleared
busy_check bsf status,rp0     ;select memory bank 1
       movlw B'00111111'     ;set port A all ip
       movwf    trisa
       bcf status,rp0
       bcf flags,0
       btfsc    portc,lcd_RS ;save RS bit in flags, 0
       bsf flags,0
       bcf portc,lcd_RS    ;access instruction register
       bsf portc,lcd_RW    ;set to read
busy_loop      bcf portc,lcd_E
       bsf portc,lcd_E
       btfsc    porta,lcd_busy ;test the busy flag, loop if still busy
       goto    busy_loop
       bcf portc,lcd_E
       bsf status,rp0    ;select memory bank 1
       movlw    B'00000000';set port A all op,
       movwf    trisa
       bcf status,rp0
       bcf portc,lcd_RS
       btfsc    flags,0 ;reinstate RS bit
       bsf portc,lcd_RS
       return
```

202

上面描述的 LCD 显示器在中小型嵌入式系统中非常有用。由于 LCD 显示器功耗较低，因此可以相对灵活地使用。但是也有不利的方面：与它交互很繁琐，即使传输一个简单的消息也需要执行一些不可省略的代码块。由于接口的信息交互速度较慢，这成为它们在高速系统中应用的一个限制因素。

8.5 与物理世界交互

不论嵌入式微控制器是否具有一个人机接口，它都必须同物理世界进行交互。因此，它必须具备检测和控制物理状态的能力。由于同物理世界交互是通过变换器进行的，因此我们主要对变换器本身进行研究。输入变换器也称为传感器，用来检测并转换物理信号到电信号。输入变换器的种类有光线传感器、温度传感器、检测物理位置的传感器如距离测量和旋转位移等。输出变换器用来将电信号转换为物理信号。例如可以引发物理运动的变换器称为执行器，这也是这里我们主要关心的。例子包括电磁线圈和电机。

我们在研究嵌入式系统时，必须要了解：可以使用什么样的变换器，它们的功能是什么，怎样与它们交互。因此，在本章后续部分会介绍一些嵌入式系统中重要的交互技术。还会介绍 Derbot AGV 中使用到的变换器。尽管选择这个例子显得有些随意，但是这些例子同其他任何例子一样都能很好的阐述接口技术——毕竟在本书中不可能涉及所有的变换器。

8.6 一些简单的传感器

目前,可使用的传感器有很多。有一些是很早以前的技术,有一些则基于当前最新的技术。包括一些"灵巧"或"智能"的传感器,它们被集成到集成电路中用于处理片上信号。这些传感器都是根据物理事件来触发物理信号到电信号的转换,有时候也可通过中间变量来触发。本节介绍的传感器包括机电的、光学的和超声的。图 8-12 展示了这些传感器。

203

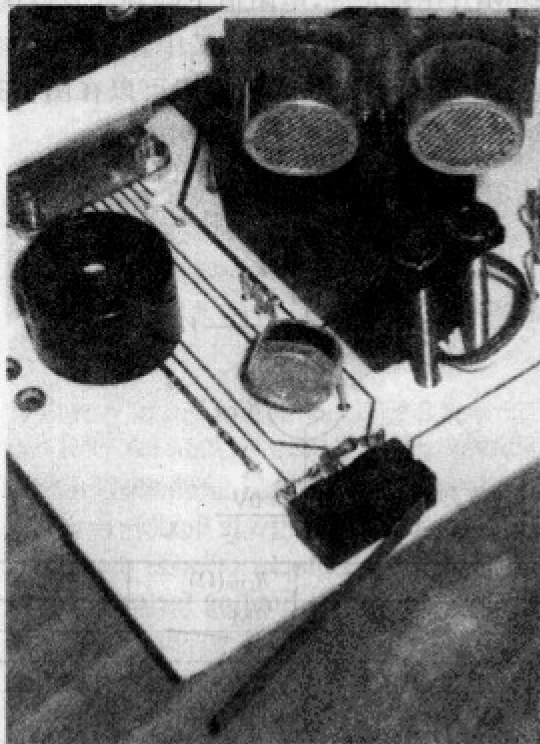


图 8-12 Derbot AGV 中的一些传感器和执行器——安放在 Futaba 伺服传动装置上的超声距离传感器,后面是光敏电阻和微开关

8.6.1 微开关

许多年来,微开关一直是进行机械位置感知的主要组成部分。在未来的若干年内,它可能还会继续占据这个重要的地位。微开关通常是单刀双掷形式,并带有连接执行器的杠杆或者滚轴。微开关有很多种,从微型的到大型的、粗糙的。大型微开关应用于重工业。从电路上来看,与微开关交互的方式同普通的转换开关是一样的,图 3-7中是一个使用微开关的例子。在工业应用中,还需要更加谨慎地设计开关电路来防止电子干扰。在 Derbot AGV 中,2 个微开关被用作"碰撞"传感器,在图 8-12 中我

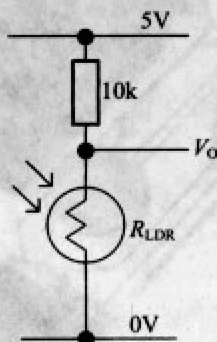
们已经看到了其中一个传感器。

8.6.2 光敏电阻

光敏电阻由一片暴露在外的半导体材料制作而成。当光线照在上面时，会在材料中产生空穴—电子对，这就提高了材料的导电性能。当光线移开，空穴—电子对会结合，导电性能则下降。光敏电阻总体效果就是，随着光线增强，光敏电阻(light-dependent resistor，LDR)的电阻值下降。

204

Derbot AGV 中使用的光敏电阻是 Silonex 公司生产的 NORP12。它在完全黑暗的条件下，电阻大约为 20MΩ；在强光下，电阻值下降到几百欧姆。它可以被用来连接成一个电压分配器，如图 8-13 所示。Derbot AGV 中有 3 个这样的光敏电阻，当 Derbot AGV 处于寻光模式时会用到它们。这些光敏电阻可以在图 A3-1 中看到。它们连接到 16F873A 的模数转换器的输入，这会在第 11 章中讲述。



光线强度	$R_{LDR}(\Omega)$	V_O
完全黑暗	2M	5.00
10	9k	2.36
1000	400	0.19

图 8-13 电压分配器中的 NORP12 光敏电阻的连接图以及示意性的电压输出值

8.6.3 光学方式的物体感知

在感知物体和物体表面时，光学方式非常有效。一种方式是当光线被打断时，传感器就感知到物体的存在；另外一种是当光线被反射回来，传感器就感知到物体的存在。许多可用的传感器都是光源和传感器封装在一起。Derbot AGV 中使用的是 Optek 公司生产的反射性光学传感器，型号为 OPB608A[8.6]。

图 8-14a 说明了这种传感器的工作原理。传感器包括一个红外发光二极管和一个光电晶体管，它们并排封装在一个塑料外壳中。这种塑料封装只允许红外线经过，其他可见光被过滤掉。如果在传感器前面适当的距离放置一个反射面，那么传感器发出的光会被反射回光电晶体管，从而导致晶体管导通。如果将传感器连接成图 8-14b 所示的电路，那么当没有反射光线返回时，电路的输出电压 V_O 为逻辑 1；当存在一个反射

面时，电压会变为逻辑 0。与传感器串联的电阻大小取决于传感器的特性，图中电阻值只是示意性的。在许多传感器中，传感器到反射面的距离非常关键，推荐距离为 3 mm 左右。

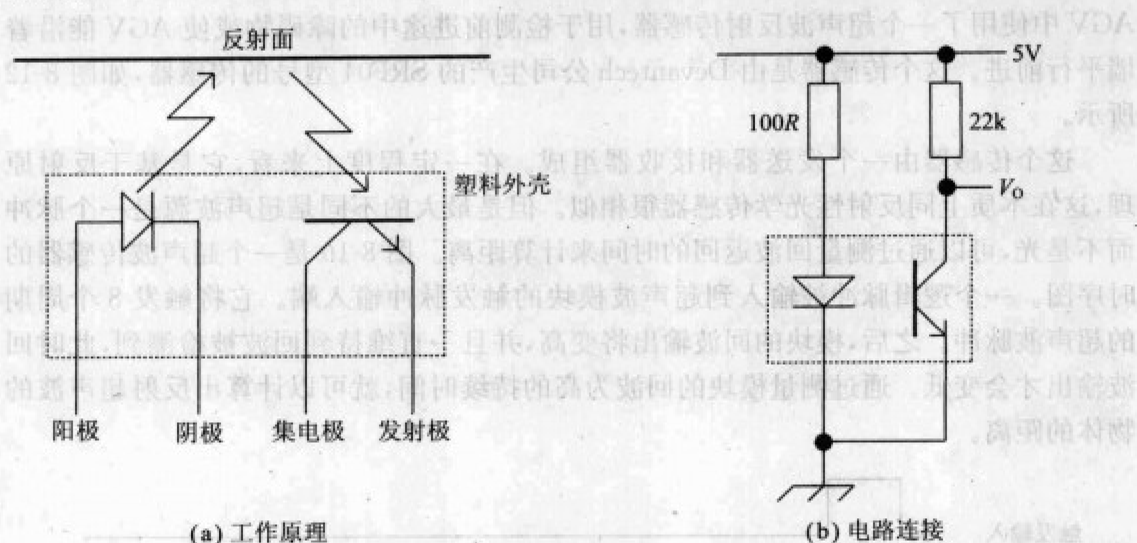


图 8-14 反射性光学传感器

8.6.4 光学传感器用于轴角编码器

上面讲述的光学传感器用来在 AGV 中产生一个非常简单的轴角编码器，如图 8-15 所示。一个黑白相间图样的卡片被固定在车轮上，在离它 3mm 的位置放置了一个光学传感器。当车轮旋转的时候，传感器将产生一个近似方形的电压波形。在卡片图样的黑色部分旋转过去之后，波形的电平会变高。（实际的波形如图 8-20 所示，在那里我们将详细研究这个信号的一些要求条件。）如果对这些脉冲进行计数，AGV 可以利用计数的结果进行距离测量和里程计算。在后续部分我们将继续深入研究计数方面的知识。这里需要注意的是，这个手工的轴角编码器对比于商业上的产品来说是相当简陋的。这是因为 AGV 中的轴角编码器每转只能产生 16 个脉冲，而商业上的产品每转可以产生 100 个脉冲，因此两者的分辨率相差很大。



图 8-15 在 AGV 中将反射性光学传感器用于轴角编码器

205
206

8.6.5 超声波方式的物体感知

超声波被广泛地用于感知和测量，比如从简单的距离测量到复杂的医学成像。AGV 中使用了一个超声波反射传感器，用于检测前进途中的障碍物或使 AGV 能沿着墙平行前进。这个传感器是由 Devantech 公司生产的 SRF04 型号的传感器，如图 8-12 所示。

这个传感器由一个发送器和接收器组成。在一定程度上来看，它是基于反射原理，这在本质上同反射性光学传感器很相似。但是最大的不同是超声波源是一个脉冲而不是光，可以通过测量回波返回的时间来计算距离。图 8-16 是一个超声波传感器的时序图。一个逻辑脉冲被输入到超声波模块的触发脉冲输入端。它将触发 8 个周期的超声波脉冲。之后，模块的回波输出将变高，并且一直维持到回波被检测到，此时回波输出才会变低。通过测量模块的回波为高的持续时间，就可以计算出反射超声波的物体的距离。

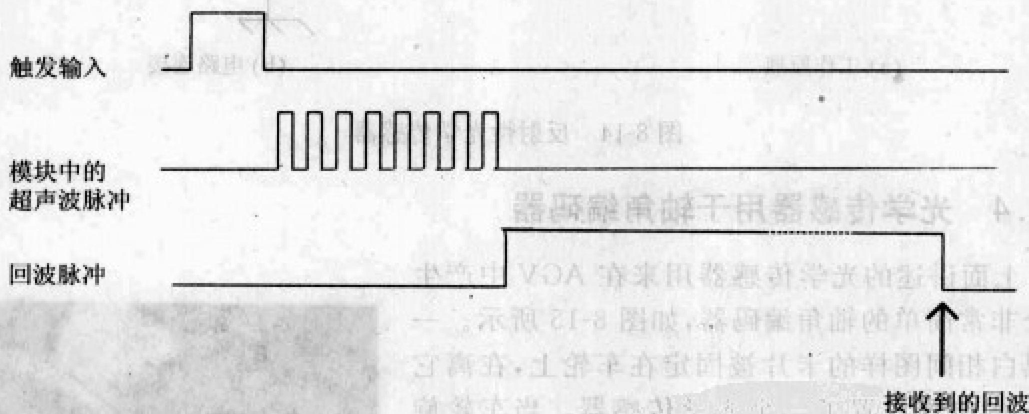


图 8-16 SRF04 超声波测距仪的简化时序图

8.7 深入学习数字信号输入

如果一个微控制器可以接收逻辑信号，那么从本质上来看，这些信号处于某个电压幅度，微控制器将这个电压幅度识别为逻辑 1，或者逻辑 0。电压幅度通常由一个逻辑系列来定义，如 TTL(Transistor Transistor Logic，晶体管-晶体管)逻辑或者 CMOS (Complementary Metal Oxide Semiconductor，互补金属氧化物半导体)逻辑。当一个器件连接到另外一个器件时，如果它们的电源电压相同并且属于相同的逻辑系列，那么通常来说，在二者之间传递逻辑电平是安全可靠的。但是，如果信号由一个非逻辑源如传感器产生，或者它们通过一个很长的通信连路传递过来，或者信号已经受到了电子干扰，那么接收器可能就不能正确地接收这些信号。

8.7.1 16F873A 的输入特性

为了确定一个信号是否能够正确地被一个逻辑器件接收，首先需要了解的是器件的输入特性。参考文献 7.1 中给出的 16F873A 端口位的输入特性如图 8-17 所示。从图中可以看到，任何在 0V～0.8V 之间的输入电压被认为是逻辑 0，在 2V～2.5V 之间的电压被认为是逻辑 1。但是这 2 个区域之间的输入电压是未定义的。

207

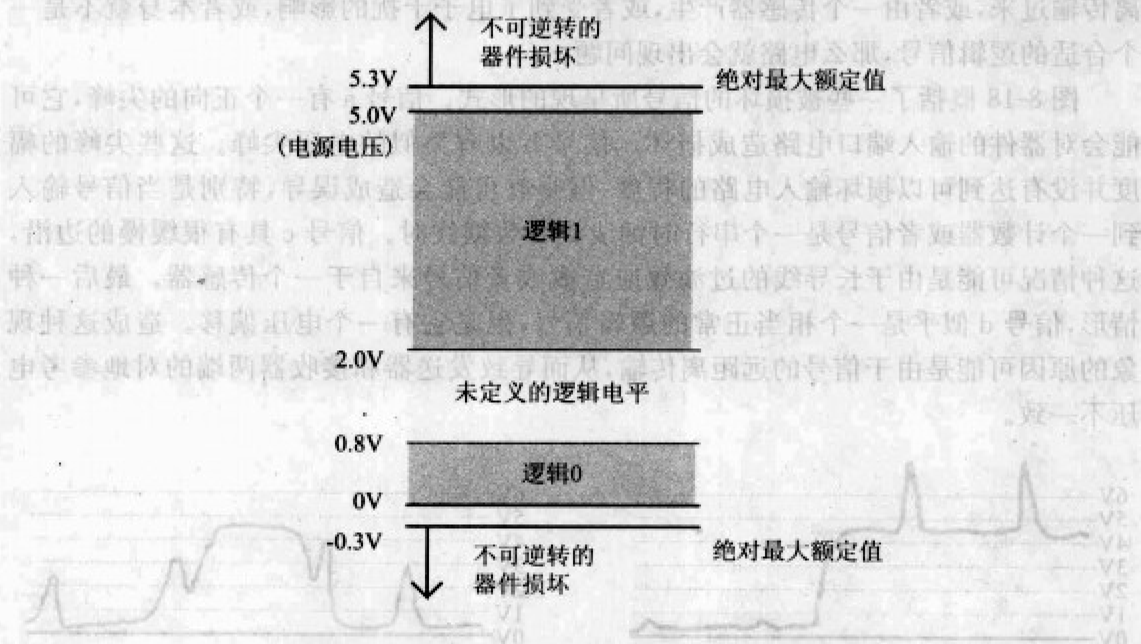


图 8-17　端口位的输入电压幅度，电源电压为 5V

如果输入电压超过了 5V，会对器件造成损坏。但是由于逻辑输入引脚几乎总是具备 2 个内部保护二极管——一个二极管从输入引脚线连接到地，一个从输入引脚线连接到电源线，这对器件提供了一定的保护。可以查看图 8-19a 中保护二极管的设计。在正常工作时，这 2 个二极管是反向偏置的。但是当输入电压高于电源电压并到达一定程度时，会使二极管导通，二极管的导通则会使输入电压固定。[①] 类似地，如果输入电压下降到 0V 以下，那么另外一个二极管将会导通。这种机制对输入电路提供了一定的保护措施。

对于 16F873A 来说，当输入电压为＋5.3V 或者－0.3V 时，输入保护二极管就会开始工作。但是这些保护二极管并不具备无限的保护能力；最大的输入电流指定为±20mA。如果输入电压严重超出绝对最大额定值，电路将会被损坏，保护二极管很有可能最先被破坏。

① 此时的输入电压等于电源电压加上二极管的导通电压。——译者注

8.7.2 确保正常的电压幅度和输入保护

现在由设计者自己确保输入电压只能稳定地处于 2 个可识别的逻辑幅度之一，即图 8-17 中的两个阴影区域之一。当然电压可以很快地通过中间未定义的区域，但是不应该停留在那里。同时电压也不能超出最大额定电压值。如果信号是由另外一个相同系列的器件在本地产生的，那么它肯定满足这些条件。但是，如果信号从很远的距离传输过来，或者由一个传感器产生，或者受到了电子干扰的影响，或者本身就不是一个合适的逻辑信号，那么电路就会出现问题。

208

图 8-18 概括了一些被损坏的信号所呈现的形式。信号 a 有一个正向的尖峰，它可能会对器件的输入端口电路造成损坏。信号 b 也有类似的电压尖峰。这些尖峰的幅度并没有达到可以损坏输入电路的程度，但是有可能会造成误导，特别是当信号输入到一个计数器或者信号是一个串行时钟线或者数据线时。信号 c 具有很缓慢的边沿，这种情况可能是由于长导线的过滤效应造成或者信号来自于一个传感器。最后一种情形，信号 d 似乎是一个相当正常的逻辑信号，但是它有一个电压偏移。造成这种现象的原因可能是由于信号的远距离传输，从而导致发送器和接收器两端的对地参考电压不一致。

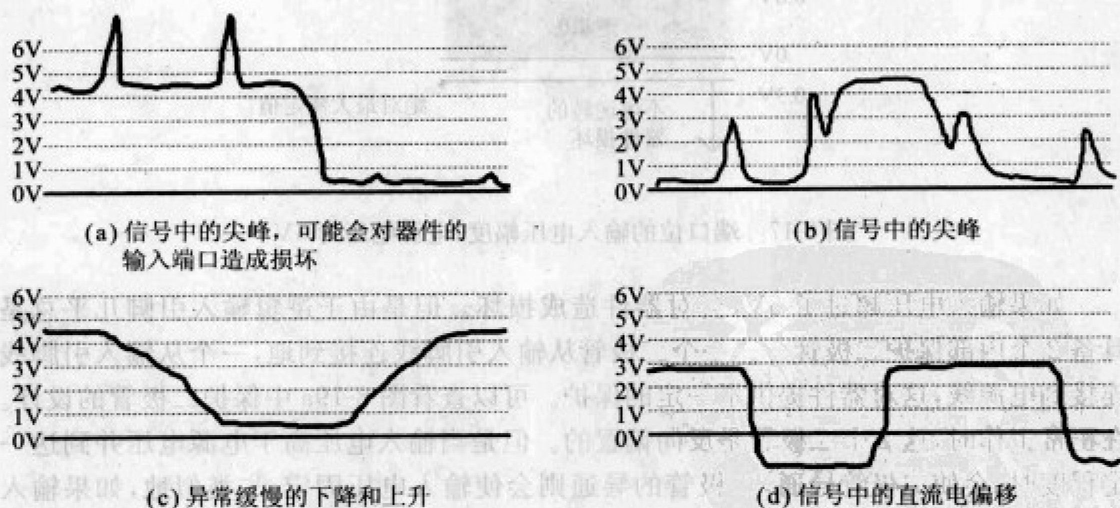


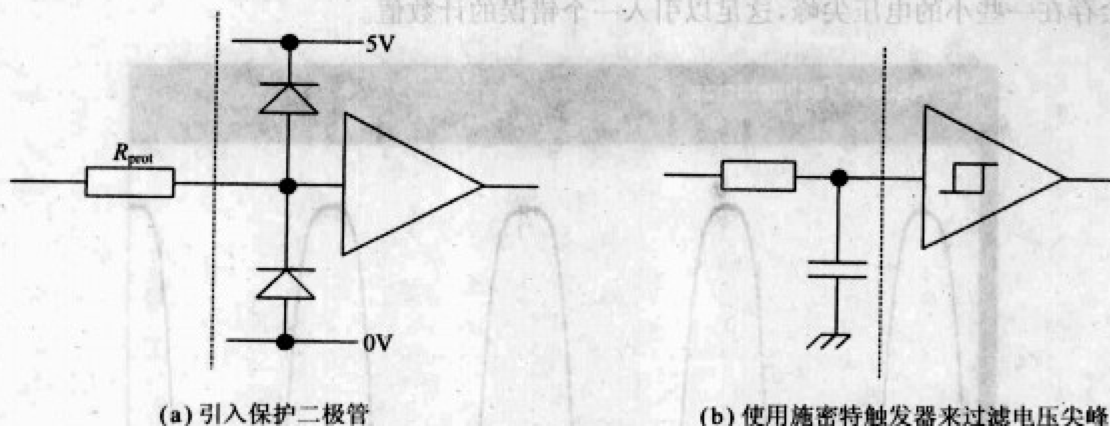
图 8-18 信号损坏的不同形式

有许多不同的技术都可以解决这些问题，以确保正常的电压幅度。这些技术可以在任何一本好的教科书中找到。图 8-19 显示了其中的 3 种技术。

1. 限制电压尖峰——使用限流电阻

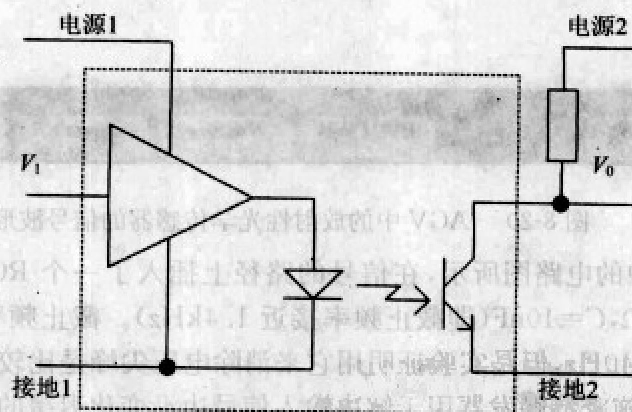
前面已经提到，如果保护二极管中流过的电流过多，它们会被烧毁。因此，如果保护二极管在实际工作时有机会导通，那么给它串联一个限流电阻是很有意义的，如图 8-19a所示。如果尖峰的幅度不是很大，这种方法可以用来解决图 8-18a 中的信号受损问题。

假设图 8-18a 中二极管最大允许电流为 20mA，R_{prot} 为 1kΩ，再假设当输入电压为 5.3V 时，上面的保护二极管将开始导通。那么最大可允许的电压尖峰值为[(20mA×1kΩ)+5.3V]，即大约 25V。



(a) 引入保护二极管

(b) 使用施密特触发器来过滤电压尖峰



(c) 使用光学隔离器进行电路隔离或幅度转换

图 8-19 一些简单的方法用来满足数字输入的条件

2. 施密特触发器

施密特触发器已经在第 3 章介绍过了。它提供了一种简单的方法使缓慢的波形边沿变得陡峭，例如解决图 8-18c 中的信号变化缓慢的问题。

3. 模拟输入信号的滤波

有时候，逻辑信号受到了电子干扰，虽然信号可能不至于损坏微控制器，但是会影响系统的正常运行。例如，如果受损的信号是计数器的输入，那么计数器的值将是一个错误的值。一个简单的 RC 滤波器（如图 8-19a 所示）有时候足以用来消除低级干扰，但是滤波器同时会使信号的边沿变得迟缓，可以再次采用施密特触发器的方法进行恢复。因此，使用滤波器的方法可以解决图 8-18a 中信号受损的问题。

210

图 8-20 的波形是由示波器跟踪 AGV 中一个发射性光学传感器获得的输出信号的波形。这些信号的连接线离电机很近，而电机是一个强大的干扰源。而且这些信号是连接到 Timer 0 和 Timer 1 的输入的。尽管信号似乎没有受到严重的破坏，但还是会存在一些小的电压尖峰，这足以引入一个错误的计数值。

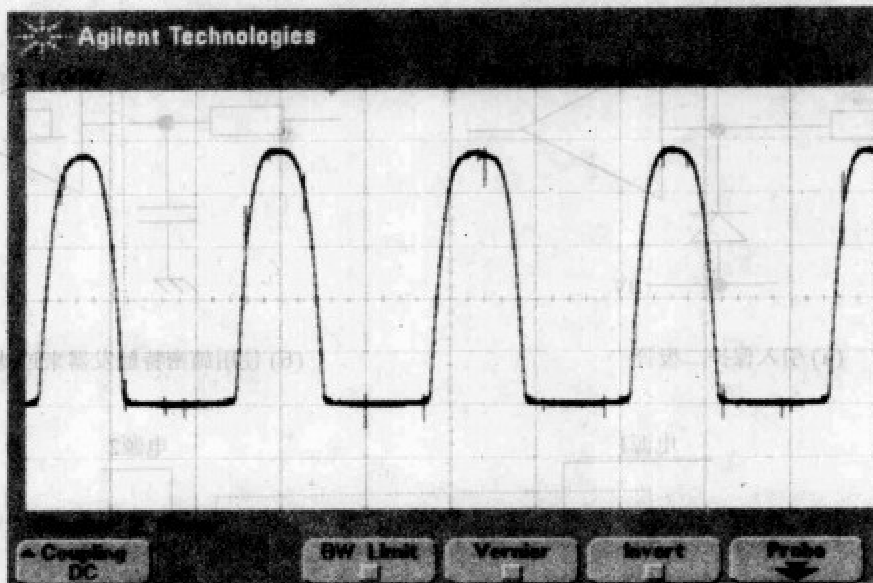


图 8-20 AGV 中的放射性光学传感器的信号波形

如图 A3-1 中的电路图所示，在信号的路径上插入了一个 RC 滤波器用来防止干扰，其中 $R=11\text{k}\Omega$，$C=10\text{nF}$(即截止频率接近 1.4kHz)。截止频率远远超过了轴角编码器的最大频率 40Hz，但是实验证明用它来消除电压尖峰是比较合适的。2 个定时器的输入端口中的施密特触发器用于解决输入信号边沿变化迟缓的问题。

1. 光学隔离器

光学隔离器(如图 8-19c 所示)是保护逻辑输入的一个有效方法，特别是当输入信号是从原距离传输过来时。输入信号驱动 LED，继而 LED 发出的光激活光电晶体管。由于输入和输出之间没有直接的电路连接，因此接地电压不一致的问题(如图 8-18d)也就解决了。

2. 数字输入滤波

考虑到信号可能会受到干扰，许多用来接收信号的集成电路都会在其电路中集成一个输入滤波器。16F873A 也采用这种做法，在第 10 章我们将会看到，一个异步串行输入电路中就采用了这种方式。简单的数字滤波策略是对输入连续抽样 3 次，然后使用一个“多数表决”电路来确定最终的逻辑值。例如，如果有 2 次 1，1 次 0，逻辑 1 将被接收，逻辑 0(可能是由干扰引起的毛刺)被丢弃掉。这种方法一定要确保抽样频率比实际信号的变化频率高。

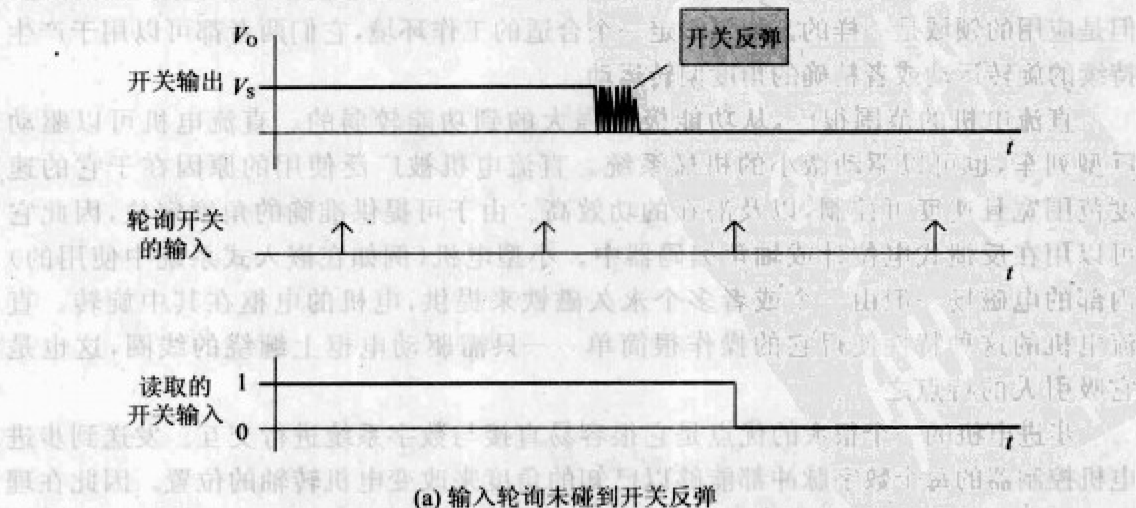
211

8.7.3 消除开关反弹

机械开关的一个很特殊的问题是当开关的接触点靠近时，开关会反弹。这会导致开关在短时间内处于开与关的状态，这段时间一般少于10ms。当打开或者关闭一个日常的电器负载，比如一盏灯，这不会是一个问题，甚至我们都不会注意到存在这种反弹。当机械开关连接到一个数字电路中，特别是频率很高的电路或者电路正在计数时，那么反弹效应的影响将是灾难性的。

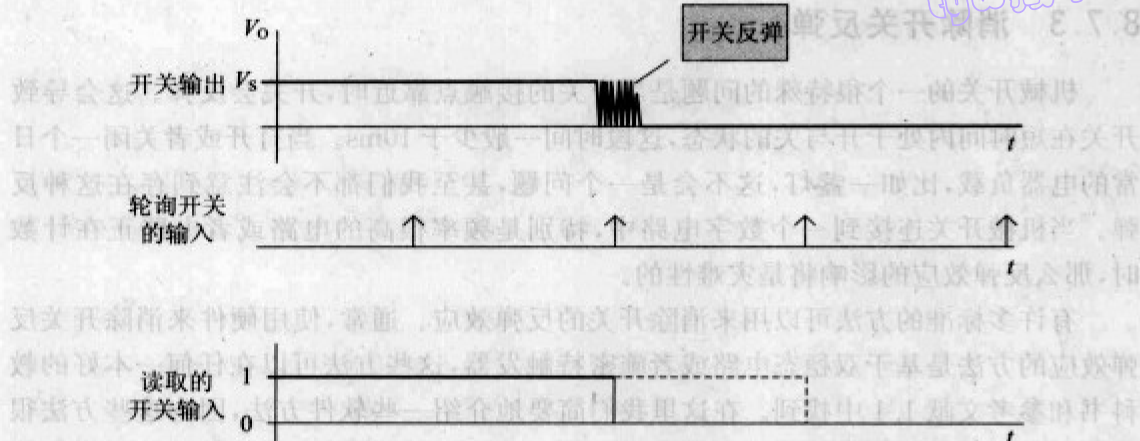
有许多标准的方法可以用来消除开关的反弹效应。通常，使用硬件来消除开关反弹效应的方法是基于双稳态电路或者施密特触发器，这些方法可以在任何一本好的教科书和参考文献1.1中找到。在这里我们简要地介绍一些软件方法，因为这些方法很有意思并且不需要额外的开销即可实现。图8-21显示了使用一种软件消除反弹方法遇到的2种情形。在图8-21a和图8-21b中，开关的输入被轮询（即周期性被读取），而且轮询的周期大于开关反弹的周期。在图8-21a中，开关的改变和反弹发生在轮询之间，因此捕获的开关状态是确定的。但是在图8-21b中，轮询发生在开关反弹时，那么读取的开关逻辑值是不确定的。不论读取的开关状态值是以前的（由点线表示的）还是新的（由实线表示的），它们都将持续一个轮询周期。因此，不管是哪种情况，开关的转换都是确定的。

有时，程序员并不想轮询开关的状态，只是希望当开关变化时引发一个中断。如果这个中断只是由一个开关来触发，并且开关总是变化到已知状态（例如，总是从高变到低），那么开关的反弹可能不是一个问题。但是如果中断可以被多个开关触发，例如本章前面介绍的键盘读取过程，那么对开关的读取可能会发生在反弹期间。在这种情况下，我们可以在检测完第一次开关变化之后使用软件方法延时一段时间。在延时之后再进行开关状态的读取，这时反弹早已结束。



(a) 输入轮询未碰到开关反弹

图8-21 通过程序轮询来消除开关反弹



(b) 输入轮询碰到开关反弹

图 8-21 (续)

8.8 执行器:电机和伺服

嵌入式系统中普遍存在的一个需求是引发物理运动。这些物理运动可能是线性的,即直线运动,也可能是旋转的。许多用于产生运动的执行器是电子器件。例如,螺线管可用于产生线性运动,"伺服"只能用于有角度的运动,直流电机或者步进电机可用于有角度的运动,也可用于旋转的运动。还有一些其他的传动方法,特别是力量强大的,包括气压和水压式。

8.8.1 直流电机和步进电机

直流电机和步进电机广泛使用于嵌入式系统中。尽管它们基于不同的工作原理,但是应用的领域是一样的。如果给定一个合适的工作环境,它们两者都可以用于产生持续的旋转运动或者精确的角度偏转运动。

212

直流电机的范围很广,从功能极其强大的到功能较弱的。直流电机可以驱动巨型列车,也可以驱动微小的机械系统。直流电机被广泛使用的原因在于它的速度范围宽且速度可控制,以及潜在的功效高。由于可提供准确的角度定位,因此它可以用在反馈式电位计或轴角编码器中。小型电机(例如在嵌入式系统中使用的)内部的电磁场一般由一个或者多个永久磁铁来提供,电机的电枢在其中旋转。直流电机的这种特性使得它的操作很简单——只需驱动电枢上缠绕的线圈,这也是它吸引人的特点之一。

步进电机的一个很大的优点是它很容易直接与数字系统进行交互。发送到步进电机控制器的每个数字脉冲都能够以已知的角度来改变电机转轴的位置。因此在理论上微处理器或者微控制器能够以不带反馈的方式来精确地控制电机转轴的速度和角坐标。但是实际上,这种理想的定位精度并不能完全实现。步进电机具有复杂的速

度特性：在一个特定的速度范围内，电机会发生机械共振；当速度较高时，则会丧失扭矩；具有有限的最高速度。所有这些都将导致电机与驱动它的数字电路失去同步。考虑到这些因素，我们认为步进电机的功效比直流电机低而且更难驱动。

213

在步进电机和直流电机两者之间做选择并不总是很容易。一般来说，如果注重精确的控制，但是对旋转运动的要求并不高，而且功耗也不是首要的考虑因素，那么通常选择步进电机比较好。如果对精确控制要求不高，但是需要较高的速度和功效，那么可以选择直流电机。

AGV需要具备可控的旋转操作来驱动车轮，即通过旋转来控制方向、速度和距离。虽然所有这些功能都可以通过步进电机来完成。但是由于直流电机的控制简单并且功率高（这对于电池供电的AGV来说是很重要的），所以我们选择使用直流电机。为了有效地实施控制，在设计中增加一些反馈是必要的。因此，我们在AGV中使用了一个简单的轴角编码器（在8.6.4节中已经讲过）。本书中的AGV版本使用的电机是由MFA/Como Drils生产的齿轮传动电机[8.8]。电机外形如图8-22所示，表A3-1和A3-2给出了电机的基本数据。

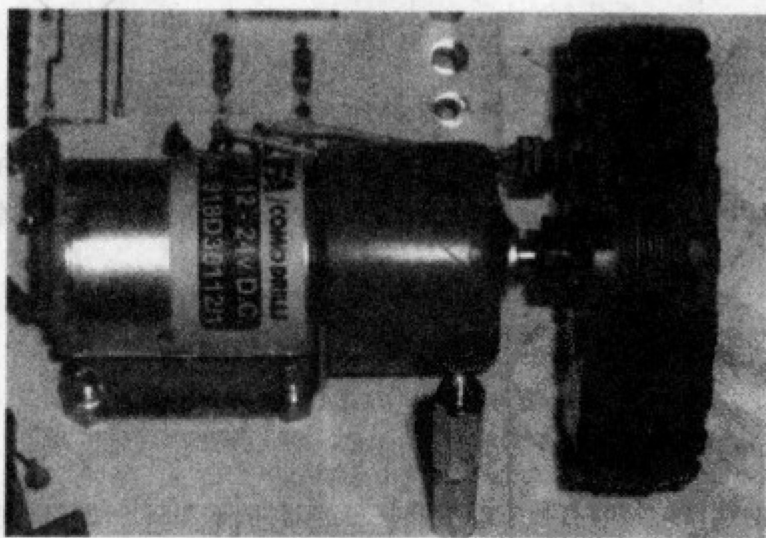


图8-22　AGV中齿轮传动电机MFA/Como RE280/1

8.8.2　角度定位：伺服传动装置

从英文单词来看，servo（伺服传动装置）是servomechanism的通用名称。由于能够进行精确的角度定位，它被广泛使用于无线电控制和机器人设计中。伺服传动装置的输出连接的是一个可以在大约180°范围内转动的转轴。它的输入一般是一个频率为50Hz（周期为20ms）的脉冲流。输入脉冲宽度确定了输出转轴转动的角度。在图8-23的例子中，1.25ms的脉冲宽度将使转轴转动0°，1.5ms时为90°，175ms时为180°。这是一个脉宽调制的例子，我们将在后续部分讲述。

214

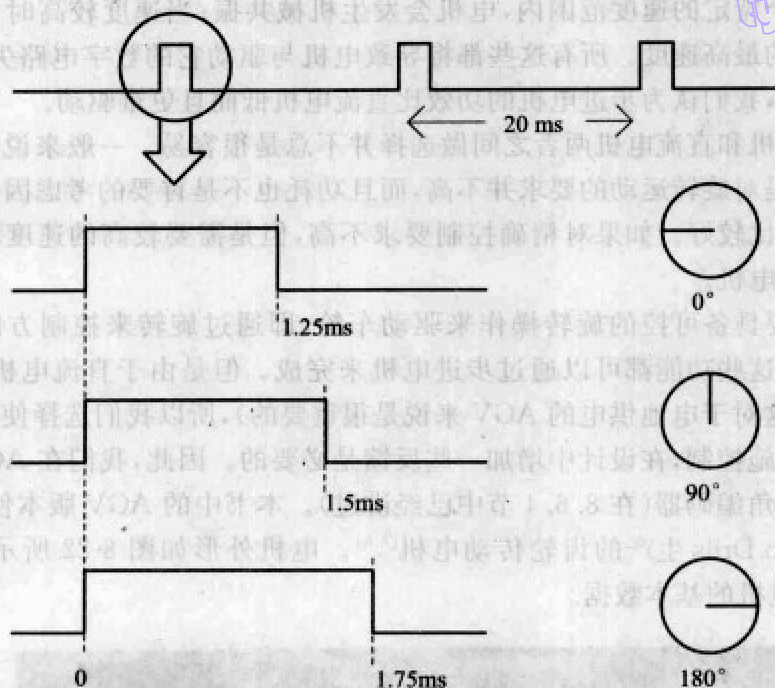


图 8-23 伺服传动装置的输入和输出特性

在不同种类的伺服传动装置中，AGV 选择使用的是 Futaba S3003 伺服传动装置[8,9]。由于可以用它来旋转超声波传感器，因此 AGV 只用一个传感器就能对不同方向进行测量。

8.9 与执行器进行交互

8.9.1 简单的直流转换

微控制器的端口位只能直接驱动负载很小的器件，如 LED。如果要驱动较大负载的器件(电流超过 10ms 或者 20ms)，或者需要驱动的器件的电源电压高于微控制器时，这些负载需要通过一个电路转换部件才能与微控制器进行交互。

晶体管开关提供了一种简单的转换直流负载的方法。图 8-24 中应用 MOSFET 和双极性晶体管这 2 种类型的晶体管进行负载转换，其中晶体管将微控制器的输出端口位或者逻辑门输出连接到电阻负载 R_L。在这 2 个电路中，当输出端口位为逻辑高时将使电流流过电阻。由于电路是开漏输出(见图 3-6)，负载的电源电压 V_S 不一定要同微控制器的电源电压相同。在很多情况下，负载的电源电压要高，例如电源电压为 5V 的微控制器可以驱动电源电压为 10V 或者 24V 的负载。

双极性晶体管(见图 8-24a)基电极上的小电流被转换成集电极上较大的电流。在 MOSFET(见图 8-24b)栅极上附加一个适当的电压以及几乎可忽略的小电流可获得较

大的漏极电流。由于 MOS 器件是完全由电压控制的，栅极可以直接连接到输出端口位。而输出电压只要满足超过栅源的阈值电压使 MOSFET 导通即可。由于 MOSFET 的连接很简单而且只要满足刚提到端口位电压超过晶体管的阈值电压这个条件即可，因此使用 MOSFET 进行微控制器周边的负载转换是一个很好的方法。

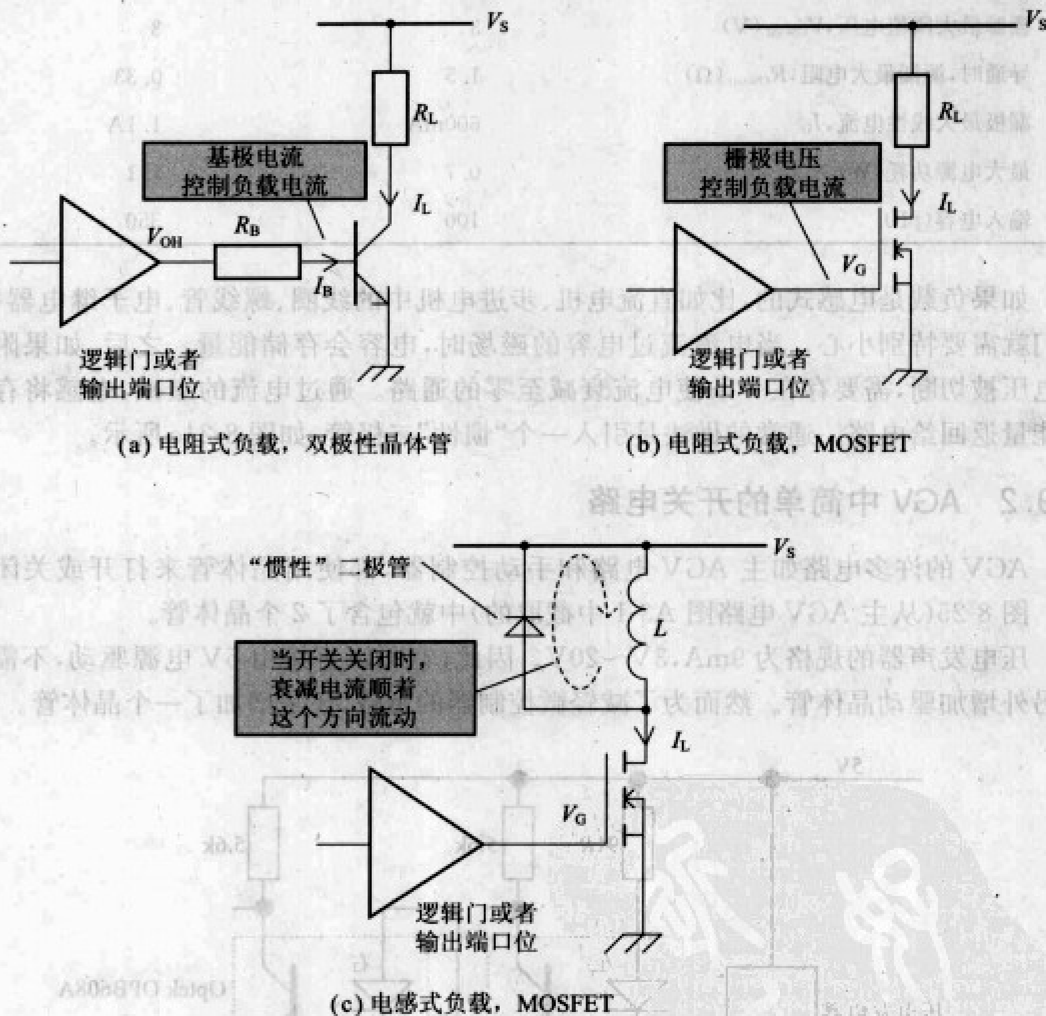


图 8-24　直流负载的晶体管转换

一些特殊型号的 MOSFET 专门用来进行逻辑电平的转换。表 8-1 为 Zetex 公司的 2 个这种类型晶体管[8,10]的例子。不管是哪个晶体管，都可以发现当栅源电压 V_{GS} 超过 3V 时，源漏之间的电阻将会下降并无限趋近一个小的数值。趋近的电阻值大小取决于晶体管的类型和内部结构。ZVN4306A 的最大"导通"电阻的数值较小，为0.33Ω。但是它的输入电容是 350pF。价格稍微便宜一点的 ZVN4206A 的"导通"电阻为 1.5Ω，但是输入电容较小，为 100pF。当驱动电路从逻辑 0 转换到逻辑 1 时，输入电容必须被驱动电路充电。对于电流—电容积较高的 MOSFET，这将成为一个必须要考虑的重要

216

因素，在很多情况下它需要自身附带驱动电路。

表 8-1 2 种通用的逻辑兼容 MOSFET 的特性

特 性	ZVN4206A	ZVN4306A
源漏最大电压，V_{DS}(V)	60	60
栅源最大阈值电压，$V_{GS(th)}$(V)	3	3
导通时，源漏最大电阻，$R_{DS(on)}$(Ω)	1.5	0.33
漏极最大线性电流，I_D	600mA	1.1A
最大电源功耗(W)	0.7	1.1
输入电容(pF)	100	350

如果负载是电感式的，比如直流电机、步进电机中的线圈、螺线管、电子继电器等，我们就需要特别小心。当电流流过电容的磁场时，电容会存储能量。之后，如果附加的电压被切断，需要存在一条使电流衰减至零的通路。通过电流的衰减，电感将存储的能量返回给电路。通常的做法是引入一个"惯性"二极管，如图 8-24c 所示。

8.9.2 AGV 中简单的开关电路

AGV 的许多电路如主 AGV 电路和手动控制器，都使用晶体管来打开或关闭负载。图 8-25(从主 AGV 电路图 A3-1 中截取的)中就包含了 2 个晶体管。

压电发声器的规格为 9mA，3V～20V。因此，它可以直接由 5V 电源驱动，不需要再另外增加驱动晶体管。然而为了减轻微控制器的负载，还是增加了一个晶体管。而

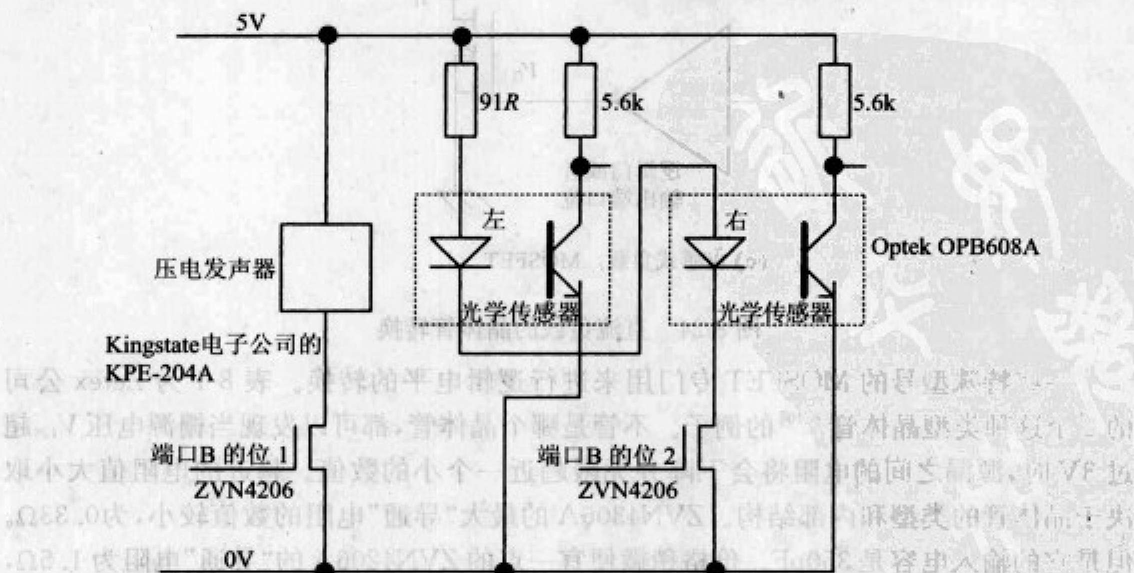


图 8-25 AGV 部分电路图—光学传感器和压电发声器

217

且由于光学传感器的功耗比较高，为了减少总的电流功耗，2 个光学传感器中的 LED 被串联在一起，如图所示。实验表明，当 LED 串联一个 91Ω 的电阻时，传感器可以正常工作。每个二极管的正向偏置电压为 1.7V(可参考器件数据)，那么电流大小为：

$$I=(5-3.4)/91$$
$$=17.6\text{mA}$$

与压电发声器一样，微控制器输出端口本来可以直接驱动传感器(尽管电流比较接近"绝对最大额定"电流 25mA)。但是，为了减轻微控制器的负载，还是使用了一个晶体管开关。

8.9.3 双向开关：H-桥

正如我们看到的，当电流固定流向一个方向时，可以很容易地打开和关闭负载。但是即使在只有单极电源电压可用的条件下，一些负载(比如直流或者步进电机)仍然需要双向电压。这种双向电压一般是通过相当巧妙的名为 H-桥的连接电路来实现，如图 8-26 所示。

在 H-桥中，2 对开关器件(一般是晶体管)连接在电源线和地之间。为了简洁起见，图 8-26 中的开关器件被表示成开关符号，并且当开关的控制输入为逻辑 1 时，开关合上。图中，开关对标示为 A 和 B。每一对都有"上端"和"下端"开关。负载连接在 2 对开关之间，整个电路形成一个 H 形。显而易见，每一对中的 2 个开关不能同时合上，否则电源和地会短路。因此，一般是通过一个逻辑反相器来驱动这 2 个开关(如图 8-26所示)，这就保证了在任何时刻只有一个开关合上。

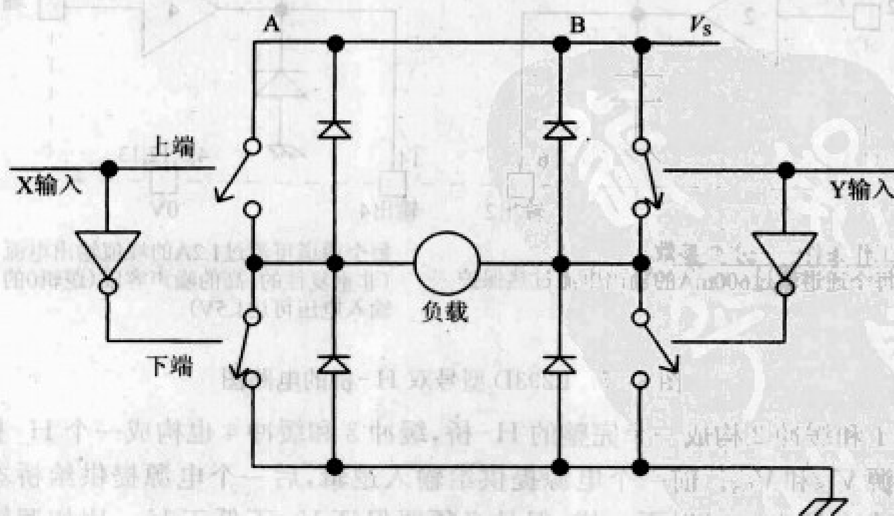


图 8-26 H-桥的工作原理

218

如果 X 输入为低，Y 输入为高，那么 A 的下端开关和 B 的上端开关将合上，其他 2 个则断开。电流会经过 B 的上端开关流过负载，再经过 A 的下端开关到达地线。如果 X 输入、Y 输入均取反，那么所有的开关状态将改变，电流将以相反的方向流动。这样

就实现了双向电流驱动。如果负载是电感式的，那么开关对必须连接一个惯性二极管，如图 8-26 所示。这种电路和它的衍生电路被应用于很多地方——从低功耗到很高功耗的应用系统中。

ST 微电子公司生产的 L293D IC 中有一个低功耗 H-桥[8.11]。它的内部包括 4 个半桥，因此可以配置成 2 个 H-桥。简化的电路如图 8-27 所示。图中每个半桥表示成一个逻辑缓冲。这种逻辑缓冲看起来似乎并不像一个半桥。我们再看一下图 8-26，当 X 输入为高时，晶体管开关对 A 的输出为高；X 为低时，输出为低。因此，半桥实际上等效于逻辑缓冲。

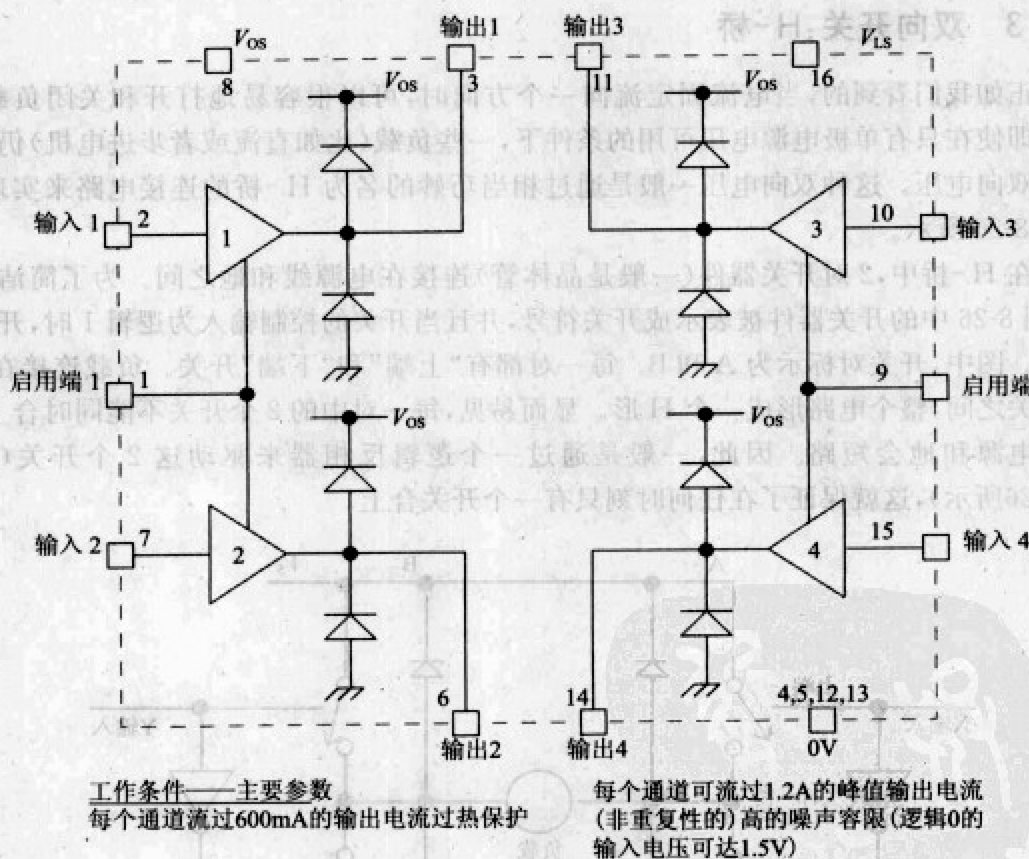


图 8-27　L293D 型号双 H-桥的电路图

缓冲 1 和缓冲 2 构成一个完整的 H-桥，缓冲 3 和缓冲 4 也构成一个 H-桥。使用了 2 个电源 V_{LS} 和 V_{OS}。前一个电源提供给输入逻辑，后一个电源提供给桥本身的电路。这 2 个电源电压可以不一样，但是必须要保证 V_{OS} 不低于 V_{LS}，比如逻辑电压为 5V，负载电压为 12V。每一对缓冲都有一个启用端，因此桥中所有的开关都能够轻易地被关掉。桥中还包含一些惯性二极管。L293D 有 4 个接地引脚，可以焊接到 PCB 板的铜面上用于提供有限的散热功能。

8.9.4 AGV中的电机开关

在AGV中是使用L293D来驱动2个电机的,详细的电路如图8-28所示。每个逻辑输入不是独立的,而是引入了一个反相器使一个输入是另一个输入的取反。这样电机只可能在一个方向上被驱动。但是每一个使能线都由一个单独的端口位来驱动,这样左右2个电机可以单独被禁用。逻辑电源连接到5V的主电源;而输出电源直接来自于输入的电池电源,即9V的电源用于驱动电机。但是由于桥内部的电压降,将输出电压降到7V左右。每一个使能线还连接了一个10kΩ的用于接地的下拉电阻,在图中没有画出(在附录3中可以看到)。使能线具有重要的作用,例如在系统初始化阶段,可以将使能线变低从而使电机失效。

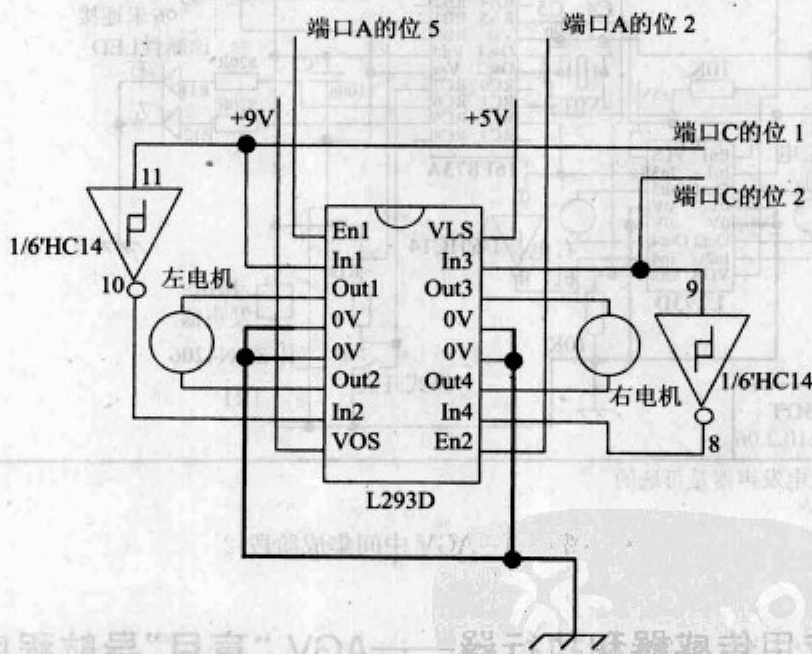


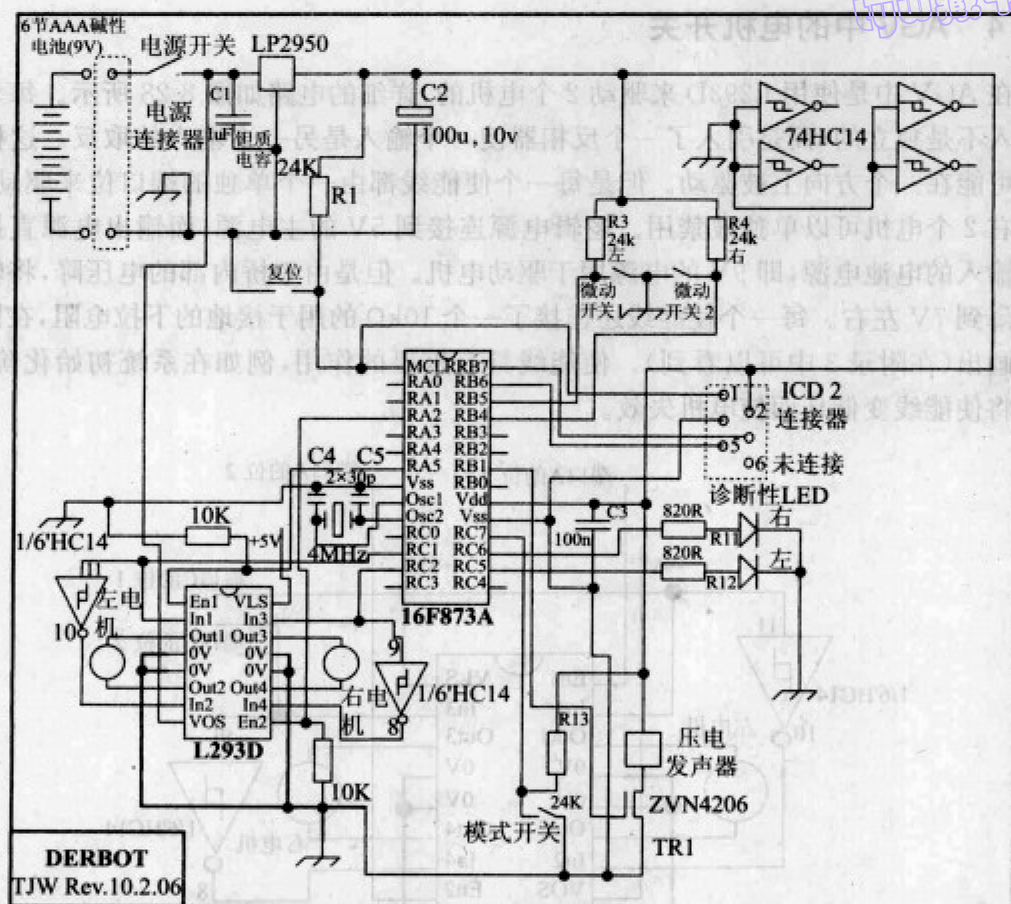
图8-28 AGV中应用L293D的电机驱动电路

8.10 AGV硬件集成

如果你正在构建AGV项目,建议你现在就增加电机和基于L293D的驱动电路,并将AGV设计成如图8-29所示的电路。如果想要让AGV可以运行,那么还需要设计一个电池盒。电池盒可以设计在一个单独的PCB板上,在它上面放置电池间隔。更进一步的构建指导可以参考本书的附属资源。

220

现在你还可以设计一个LED或者LCD版本的手动控制器卡。然后将本章第一部分给出的程序下载到手动控制器中对手动控制器进行测试。程序中手动控制器没有与主AGV进行交互,但是在后续部分我们会涉及这些内容。



221

图 8-29 AGV 中间集成阶段 2

8.11 应用传感器和执行器——AGV“盲目”导航程序

当构建了上面所述的 AGV 项目后,就可以在这个硬件平台上运行本书附属资源中的程序 **Dbt_blind_Nav**。这个程序主要特性如例程 8-4 所示。程序将使 AGV 导向车向前行驶,直到车前面的微开关检测到障碍。此时,AGV 后退并调整方向,然后继续在新的方向上前进。

主程序是从标号为 **start** 处开始运行的,它首先设置左右电机向前行驶。这本来是可以通过简单地将电机打开来实现的。但是,由于电机的默认速度较高,超过了这个很简单的应用程序的需求。因此,使用脉宽调制来设置一个较慢的速度。脉宽调制的具体过程在子例程 **leftmot_fwd** 和 **rtmot_fwd** 中,程序依次调用这些子例程。下面的例程 8-4 中没有包括这些程序的细节,但是可以在全书的程序清单中找到,在后续部分我们会仔细研究它们。然后,程序进入标号为 **loop** 的循环。在循环中,程序重复测试前

面2个微动开关，AGV继续行驶直到其中一个微动开关检测到障碍物。当一个微动开关检测到障碍物时，AGV会停下来，然后压电发声器发声，程序将再次调用脉宽调制子例程使AGV倒退1.5s左右。然后，AGV通过一个电机的前进和另一个电机的后退来调整方向。之后，程序返回到主程序的循环开始处，AGV继续向前行驶。

例程 8-4 AGV 盲目导航程序(部分)

```
;*******************************************************************
;Dbt_blind_Nav
;Derbot moves by "blind" navigation.
;Moves forward, and reverses and turns on bump.
;Fixed rate PWM applied to set reasonable speeds.
;
;TJW 5.5.05                                    Tested 9.5.05
;*******************************************************************
...
(Memory Allocation and Initialisation omitted)
...
;start motors
start call leftmot_fwd      ;sets left motor running forward
      call rtmot_fwd        ;sets right motor running forward
;test for bumps - reverse and turn if either microswitch closes
loop btfss  portb,us_rt     ;test right microswitch
      goto  rev_rt
      btfss portb,us_left   ;test left microswitch
      goto  rev_left
      call  delay100
      goto  loop
;
rev_rt bsf  portc,led_rt
      bcf   porta,mot_en_left ;stop motors
      bcf   porta,mot_en_rt
      bsf portb,sounder     ;small bleep from sounder
      call delay200
      bcf portb,sounder
;reverse both motors
      call leftmot_rev
      call rtmot_rev
      call delay500
      call delay500
      call delay500
      call leftmot_fwd       ;left motor forward to turn
      call delay500
      call delay500
      bcf   portc,led_rt
      goto start
;
rev_left bsf portc,led_left
      bcf   porta,mot_en_rt ;stop motors
      bcf   porta,mot_en_left
      bsf portb,sounder   ;small bleep from sounder
      call delay200
      bcf portb,sounder
```

```
;reverse both motors
      call leftmot_rev
      call rtmot_rev
      call delay500
      call delay500
      call delay500
      call rtmot_fwd        ;right motor forward to turn
      call delay500
      call delay500
      bcf   portc,led_left
      goto start
...
(subroutines omitted)
...
```

小结

- ☐一个嵌入式微控制器必须具备同物理世界和与人交互的能力。
- ☐许多人机交互可通过开关、键盘、显示器件来完成。
- ☐为了与物理世界交互，微控制器必须要与一些变换器进行交互。嵌入式系统的设计者需要了解一些主要的可用传感器和执行器，必须时刻准备与该领域的最新技术保持同步。
- ☐与传感器的交互需要对信号条件技术进行深入理解。
- ☐与执行器的交互需要对电源转换技术进行深入理解。

参考文献

223

8.1. 12.7 mm (0.5 inch) Single Digit Numeric Displays (2003). Kingbright, DSAD0006, http://www.kingbright.com.tw

8.2. HD44780 Data.可从许多网站获取，比如 http://www.electronic-engineering.ch/microchip/index.html

8.3. Interfacing PICmicro MCUs to an LCD Module (1997). Microchip Technology Inc., DS00587B.

8.4. Powertip; http://www.powertip.com.tw/

8.5. Silonex; http://www1.silonex.com/

8.6. Optek; http://www.optekinc.com/

8.7. Devantech. SRF04 数据可从 http://www.robot-electronics.co.uk/和其他一些网站获取。

8.8. MFA/Como Drills; http://www.comodrills.com/

8.9. Futaba; http://www.futaba-rc.com/

8.10. Zetex; http://www.zetex.com/

224

8.11. STMicroelectronics; http://www.st.com/stonline/

第 9 章
深入学习计时

在第 6 章中，我们了解到在嵌入式环境中计数和计时是非常重要的，也知道了数字计数和将数字计数用于时间计算是非常方便的。现在，我们需要深入地学习这一功能，特别是在时序领域中。一旦有一些很好的时序工具，就可以使用它们来巩固和增强微控制器的其他功能，例如产生串行数据或脉宽调制。还可以使用它们来简化复杂的外部活动，例如为一个引擎管理系统产生时序信号。

本章将讨论嵌入式环境中计数和计时的需求，并且在较为复杂的层次上开发满足这些需求的功能。计数仍然是基本技术，而计时功能是它最突出的能力。

本章的一个重要主题是讨论增强型的计数器/定时器的结构。这将使微控制器能够将大量基于时间的操作移交给增强型的计数器/定时器硬件。那么，硬件在处理与时间相关的功能时，程序可以继续在微控制器上执行。基于时间的操作包括：

- ❑ 维持持续的计数功能；
- ❑ 在定时器硬件中记录（捕捉）事件发生的时间；
- ❑ 使用定时器硬件在特定时刻触发事件；
- ❑ 配置硬件以产生重复的基于时间的事件；
- ❑ 测量频率以及由频率表示的物理变量，例如电机速度。

本章所讨论的结构为 16F873A 微控制器的计数器/定时器结构。学习这些内容既可以增长我们使用 16F873A 器件以及与它相近器件的专业技术知识，又可以了解适用于任何一个微控制器系统的计数器/定时器的基本概念。本章通过 AGV 的一些例子来介绍使用计数器/定时器计时的各种功能，最后一节概括性地介绍了 AGV 中定时/计数功能必需的硬件构成。

值得注意的是，Microchip 公司在涉及他们的计数器/定时器模块时，倾向于使用术语“定时器”。在本章中，“定时器”和“计数器/定时器”有时交换使用。

9.1 深入学习计数和计时

在第 6 章的基础上，我们需要继续学习计数和计时的更高级的应用，如上面所列出的应用。这些更高级的应用必须扩展基本的计数器/定时器硬件。所以，本章在介绍每个新的计数或计时技术的同时，也描述了该技术需要的硬件。

在 6.3 节介绍了 PIC® 16 系列的 Timer 0。如果你忘记了 Timer 0 的结构，花点时间再看看图 6-8，回忆一下它的基本特征。Timer 0 的核心是一个数字计数器，该计数

器映射到存储器中，并且可读可写。计数器的输入时钟可以从 2 个输入源中选择。当选择外部输入时钟时，该模块通常被认为处于“计数器”模式。另外，当把内部振荡器信号连接到计数器时，该模块被认为处于“定时器”模式。模块中使用了一个预分频器，用来对输入的时钟信号进行分频。当计数到它的最大值时，计数器就会溢出，计数值回到零，并且产生一个中断。

PIC 16F873A 有 3 个定时器，它们有一些重要的附属功能，现在我们将讨论这 3 个定时器。前面我们刚刚回顾了相对简单的计数器/定时器结构，它是我们接下来进行更高级定时器设计的基础。

9.2 16F87XA Timer 0 和 Timer 1

9.2.1 Timer 0

16F873A 使用的定时器是标准的中端型号微控制器的 Timer 0 模块。所以，它的计数器/定时器的设计与 16F84A 相同。6.3.3 节已经详细描述了 Timer 0，这里不再赘述。

9.2.2 Timer 1

PIC 中端型号的 Timer 0 的位宽限制为 8 位。Timer 1 是直接在 Timer 0 的结构基础上构成的，但还是有许多重要不同之处，如图 9-1 所示。首先，它的位宽是 16 位的，由 2 个 8 位寄存器 **TMR1H** 和 **TMR1L** 组成，如图所示。这 2 个寄存器都是特殊功能寄存器(Special Function Register，SFR)，通常是可读可写的。在图 7-6 的寄存器文件映射中可以找到它们。2 个寄存器合在一起可以计数的范围为 0000～$FFFF_H$（即 65536_D）。当计数从 $FFFF_H$ 回到 0 时，中断请求标志 **TMR1IF**(见图 7-10 的中断结构框图)被设置。通过启用该标志来触发一个溢出中断。

226

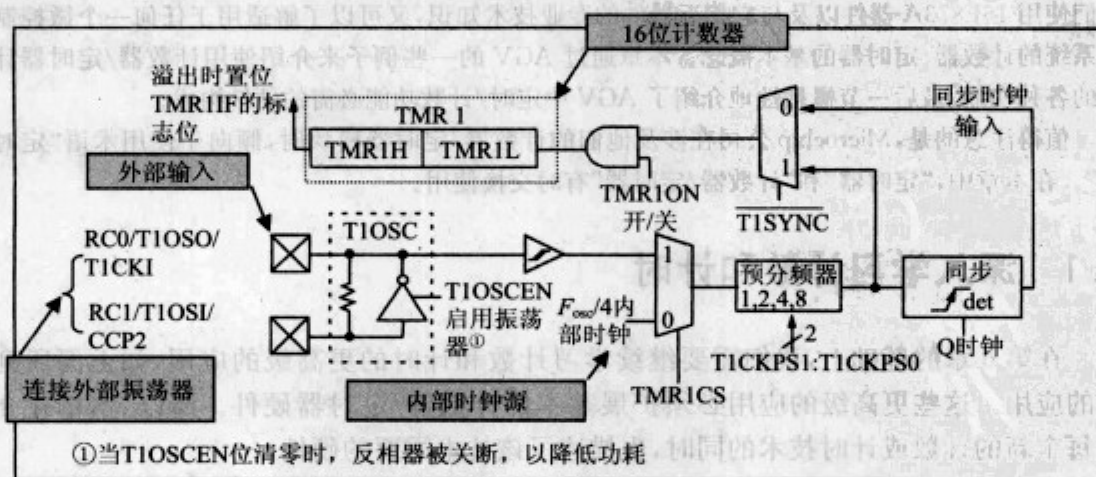


图 9-1 16F87XA Timer 1 框图(阴影框中所附标签为作者所加)

通过 T1CON 寄存器来控制 Timer 1,如图 9-2 所示。通过 TMR1ON 位来开启和关闭定时器。定时器有 3 个不同的时钟源,在图 9-1 中可以找到。对于计数功能,定时器必须使用外部输入时钟 T1CKI,它和端口 C 共享引脚 0。对于计时功能,在电路中仍然可以选择内部时钟振荡器 $F_{osc}/4$。通过控制寄存器中的 TMR1CS 位来选择是使用外部时钟还是内部时钟。通过配置专用的 Timer 1 外部振荡器,定时器可以获得第 3 个时钟源,专用的 Timer 1 外部振荡器连接到图 9-1 中所示的 2 个外部引脚。这可以消除计时器对主振荡器频率的依赖。外部振荡器的频率可以和主振荡器的频率完全不同,并且在休眠模式下主振荡器关闭后定时器仍然可以在外部振荡器下运行。通过 T1OSCEN 位可以启用外部振荡器。Timer 1 的振荡器输入同主 LP 振荡器相同(参见 3.5.3节)。外部振荡器的频率较低,最高约为 200kHz。通常使用 32.768kHz 的晶体,然后通过分频得到一个二分频的时基。

U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	T1CKPS1	T1CKPS0	T1OSCEN	T1SYNC	TMR1CS	TMR1ON
位 7							位 0

位 7~6　未实现:读作"0"

位 5~4　T1CKPS1:T1CKPS0:Timer 1输入时钟预分频比选择位
11 = 1:8 预分频比
10 = 1:4 预分频比
01 = 1:2 预分频比
00 = 1:1 预分频比

位 3　T1OSCEN:Timer 1振荡器使能位
1 = 振荡器启用
0 = 振荡器关闭(振荡器的反相器被关断,以降低功耗)

位 2　T1SYNC:Timer 1外部时钟输入同步控制位
当TMR1CS = 1时:
1 = 不同步外部时钟
0= 同步外部时钟
当 TMR1CS = 0时:
此位被忽略。当TMR1CS = 0时Timer 1使用内部时钟

位 1　TMR1CS:Timer 1时钟源选择位
1 = 来自RC0/T1OSO/T1CKI引脚的外部时钟(上升沿计数)
0 =内部时钟($F_{osc}/4$)

位 0　TMR1ON:Timer 1启用位
1 = 启用Timer 1
0 = 禁用Timer 1

注:在该型号的18系列寄存器中,位 7被称为RD16。如果将位 7置1,它将启用"16位读/写"模式

图 9-2　Timer 1 的控制寄存器 T1CON(地址为 10_H)

无论选择哪个时钟源,电路都会设置预分频器。预分频器仅有 2 个控制位 T1CKPS1 和 T1CKPS0,只能得到 3 个有效的分频值 2、4 和 8,因此它所提供的分频范围没有 Timer 0 的宽。最后,通过位$\overline{\text{T1SYNC}}$对外部输入进行同步。如果使用捕捉或比较模式(在后续各节会介绍),定时器必须同步工作。但是,如果定时器在异步方式下工作,当微控制器处于休眠模式时,定时器可以继续工作。

当选择外部时钟源时,计数器总是在时钟上升沿时递增(在 Timer 0 中,可以选择是在上升沿递增还是在下降沿递增)。但是,计数器在开始计数前必须首先在下降沿

被触发。因此，这会非常危险，有可能丢掉第一个计数脉冲。在后面的 AGV 程序中我们将学习如何克服这个小问题。

9.2.3 使用 Timer 0 和 Timer 1 作为 AGV 里程表的计数器

之前已经指出计时是本章最重要的主题，因此第一个例子就是关于计数的。车辆的一个基本需求是能够计算出它行驶的距离，即通常所说的里程。在 AGV 中这是通过一个手工制作的光学轴角编码器来完成的，8.6.4 节已经对它进行了讲述。通过对轴角编码器产生的脉冲进行计数并且计算出每个脉冲所表示的车轮转过的几何距离，就可以对行驶的总里程进行测量。

例程 9-1 应用轴角编码器实现了 AGV 的一个简单里程表。如图 A3-1 所示，2 个光学传感器的输出连接到 AGV 中微控制器的 Timer 0 和 Timer 1 的输入端。这 2 个定时器都配置成计数器模式。对于该原型中使用的车轮直径以及附录 3 中描述的轴角编码器的应用，程序将驱动 AGV 向前行驶 1m 的路程。然后，它立刻完成一个 180° 的转弯，之后再次向前行驶 1m。程序以这种方式持续循环。

例程 9-1 AGV 中里程表的应用

```
;**************************************************************
;odometry_test
;Runs forward a fixed distance, turns by 180 degrees,
;and returns - looping continuously.
;
;TJW 19.5.05                                Tested 20.5.06
;**************************************************************
...
(comments, and memory definition omitted)
...
     org 00
;set up SFRs in Bank 1
...
(set up Tris A, B, C)
...
     movlw  B'01000100'
     movwf  adcon1 ;set port A for right analog/digital mix
     movlw  B'11101000' ;set up Timer 0: external input, low to high
                                         ;transition,no prescale
     movwf  option_reg
     movlw  D'250'      ;set PWM prd
     movwf  pr2
;set up SFRs in Bank 0
     bcf    status,rp0   ;select bank 0
     movlw  B'00000011' ;set up Timer 1: no prescale, oscillator
     movwf  t1con           ;disabled, external sync input
...
further initialisation, and opening section of program
...
;*************************************************
;run forward fixed distance, then turn and return
;*************************************************
```

```
opto_move clrf tmr0           ;clear timers
       clrf   tmr1l
       clrf   tmr1h
       clrf   flags
       btfss  portc,0         ;increment T1 if ip is zero, as first rising edge
                                                            ;isn't detected
       incf   tmr1l
       call   leftmot_fwd     ;start motors running
       call   rtmot_fwd
opto_loop call opto_to_led ;transfer opto states to diagnostic leds
;move forward set distance (1m)
       movlw  D'91'           ;test if counter has reached this value
       subwf  tmr0,0
       btfsc  status,z
       bcf    porta,mot_en_left  ;disable motor if value reached
       movlw  D'91'
       subwf  tmr1l,0
       btfsc  status,z
       bcf    porta,mot_en_rt
;if both motors stopped, proceed to turn, otherwise loop
       btfsc  porta,mot_en_left
       goto   opto_loop
       btfsc  porta,mot_en_rt
       goto   opto_loop
;now turn
       call   delay500        ;ensure AGV is at rest
       movlw  00
       movwf  tmr0            ;clear timers and flags
       movwf  tmr1l
       btfss  portc,0         ;increment T1 if it is zero,
       incf   tmr1l
       call   leftmot_fwd     ;turn on spot, left motor forward, right back
       call   rtmot_rev
;execute the turn
opt_loop1 call opto_to_led ;transfer opto states to diagnostic leds
;rotate by 180 degrees
       movlw  D'23'           ;test if counter has reached this value
       subwf  tmr0,0
       btfsc  status,z
       bcf    porta,mot_en_left ;disable motor if value reached
       movlw  D'23'
       subwf  tmr1l,0
       btfsc  status,z
       bcf    porta,mot_en_rt
;if both motors stopped, proceed to straight line, otherwise loop
       btfsc  porta,mot_en_left
       goto   opt_loop1
       btfsc  porta,mot_en_rt
       goto   opt_loop1
       call   delay500        ;ensure we're at rest
       goto   opto_move
```

228
~
229

```
;*****************************************************
;SUBROUTINES
;*****************************************************
...
motor control and delay subroutines
...
;transfers opto sensor state to leds
opto_to_led bcf portc,led_left    ;preclear left led
      btfss  porta,4
      bsf    portc,led_left      ;but set it if opto on
      bcf    portc,led_rt  ;preclear right led
      btfss portc,0
bsf    portc,led_rt   ;but set it if opto on
return
end
```

230

与前面一样，为了减小篇幅，书中没有列出完整的程序清单，但是可以在本书的附属资源中找到它。这个程序中的许多特性都很有启发性。程序以标准的方式来设置端口位，这种设置方式是 AGV 中使用端口位的一般方法。**ADCON1** 寄存器用来控制端口 A 中的位是用于模拟输入还是数字输入/输出。在这里，端口 A 被配置成部分模拟部分数字(这也是最终 AGV 硬件所需要的配置)，即使这个程序没有使用模拟输入。

通过回顾图 6-9 的 **OPTION** 寄存器，可以验证源程序中的注释对 Timer 0 设置的描述。类似地，查看图 9-2 可以验证 Timer 1 的设置。

在程序段 **opto_move** 开始处，2 个定时器都被清零。然后，程序测试 Timer 1 的外部输入。如果为 0，计数值加 1。这是由于硬件没有检测到输入信号的第一个上升沿。然后程序使用例程 8-4 中相同的子例程设置 2 个电机向前行驶。随后，程序运行到 **opto_loop**循环处。在这里，计数值连续被测试。对于一个每转产生 16 个脉冲的黑白相间的轴角编码器来说(在附录 3 中描述)，计数值为 91 时表示 AGV 行驶了 1m 的路程。当计数器到达 91 时，电机各自的使能位被清零，电机停止前进。

当 2 个电机都停止时，AGV 立即通过驱使 2 个电机向相反的方向前进，来完成转弯。如果电机要转过 180°，那么每个车轮必须覆盖的距离可以通过几何关系来计算得到(附录 4 中介绍)。通过计算，这个距离等价于程序中定时器的计数值 23。在程序运行时，每个车轮在转弯过程中滑过的弧度通过轴角编码器来测量。当车轮覆盖的距离完成时，电机停止。然后 AGV 行驶相同的固定距离重新回到起点，之后再一次继续处理转弯过程。

这个程序演示了计数在里程表中的应用，同时也揭露了它的缺点。使用简单的自制轴角编码器进行距离测量的精度是很低的，只适用于演示——尽管这种方法很有趣。AGV 只能完成近似的 180°转弯以及近似回到原点。如果它持续来回行驶，返回点将离最初的起点越来越远。这是由不断累积的误差所导致的。并且，程序中没有对电机的速度进行控制。尽管由同一个电源驱动，但是任何 2 个电机完全相同的机率很

小。因此，AGV 逐渐趋于曲线前进，尽管很轻微。当本章后面实现了速度测量之后，就可以解决这个问题了。第 11 章会讲述速度控制。

9.2.4 使用 Timer 0 和 Timer 1 产生重复性中断

计数器/定时器的一个重要应用是经常使用它来产生连续的间断触发的一系列准确定时中断。这些连续中断可用于主机的许多计时应用，例如它是形成实时操作系统(Real Time Operating System，RTOS)这个复杂编程框架的基础，这是第 18 章的主题。这一系列中断有时被称为时钟滴答(clock click)。不要把它同我们已经讲述的时钟振荡器的概念混淆了。这个时钟滴答与时钟振荡器本身一样非常基础，它是许多时基操作的基础。

Timer 0 和 Timer 1 可以很容易地用来产生这个时钟滴答，因为它们都可以触发一个溢出中断。图 9-3 说明了它的工作原理。在定时器模式和中断使能配置下定时器连续运行。图中，定时器值由一个阶梯波形来表示。它重复计数到最大值，然后溢出，之后返回至零。每一次溢出，都将触发一个中断。

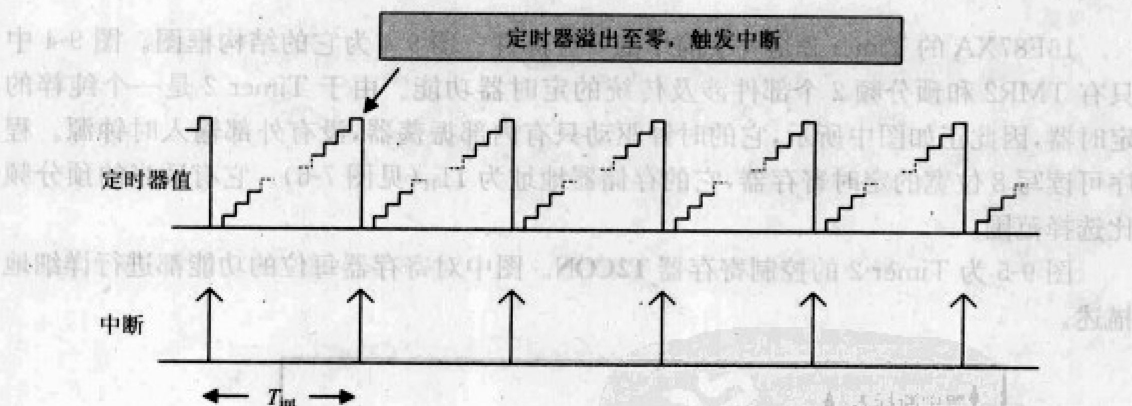


图 9-3 产生“时钟滴答”——一个重复乏味的中断流

231

一般来说，如果定时器是 n 位的，那么定时器的值从 0 变到下一次 0 需要 2^n 个周期。那么每个中断的间隔 T_{int}，可计算如下：

$$T_{int} = 2^n \times t_{clk}$$

其中 t_{clk} 是定时器输入时钟的周期。

设计实例 9-1

假定振荡器的频率为 4MHz，通过 Timer 0 和 Timer 1 可产生的最慢的“时钟滴答”频率为多少？在这种假设下，这 2 个定时器的次低频率又是多少？

Timer 0 的结构可以参考图 6-8 和图 6-9。为了获得最慢的时钟频率，预分频比设置为÷256。因此，输入到定时器本身的时钟频率为 1MHz/256，即 3.906kHz。这个 8 位计数器本身的动作又将这个频率除以 256 用来产生时钟滴答的频率，这个频率为

3.906kHz/256，即 15.26Hz。如果将预分频比设置为÷128，就可以产生次低频率，为 30.52Hz。

Timer 1 的结构可以参考图 9-2 和图 9-3 所示。Timer 1 的最大预分频比是÷8。当设置为最大分频比时，输入到定时器本身的时钟频率为 125kHz。计数器的动作又将这个频率除以 2^{16}，这将产生一个 1.91Hz 的中断频率。如果将预分频比设置为÷4，将产生次低的中断频率，即 3.81Hz。

通过这个实例，我们发现了一个很有趣的事实：这 2 个定时器最低频率实际上相差不大。尽管 Timer 0 只有 8 位，但是它有一个非常有效的预分频器来弥补定时器位宽小的缺陷，于是同样也可以产生较低的中断频率。

一个溢出中断的例子可以在例程 9-3 中看到。

9.3 16F87XA 的 Timer 2、比较器和 PR 2 寄存器

9.3.1 Timer 2

16F87XA 的 Timer 2 是一个简单的 8 位部件。图 9-4 为它的结构框图。图 9-4 中只有 TMR2 和预分频 2 个部件涉及传统的定时器功能。由于 Timer 2 是一个纯粹的定时器，因此正如图中所示，它的时钟驱动只有内部振荡器，没有外部输入时钟源。程序可读写 8 位宽的定时寄存器，它的存储器地址为 11_H（见图 7-6）。它有适当的预分频比选择范围。

图 9-5 为 Timer 2 的控制寄存器 **T2CON**。图中对寄存器每位的功能都进行详细地描述。

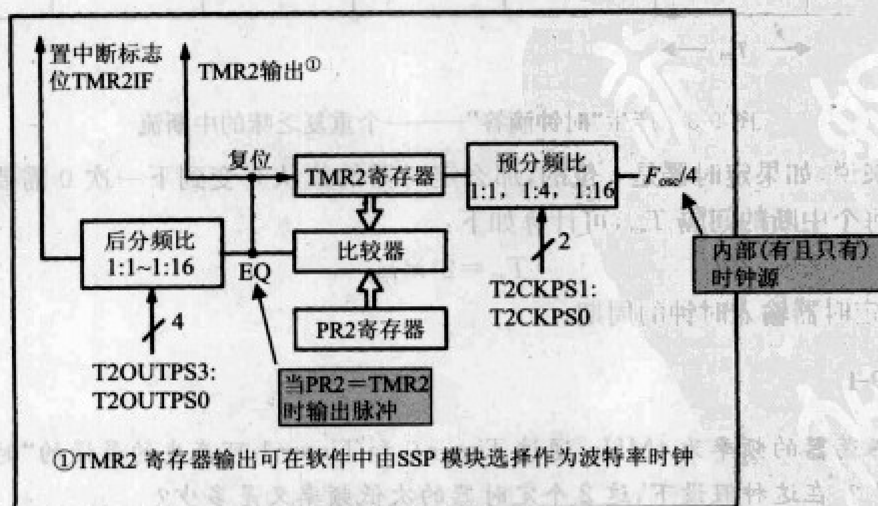
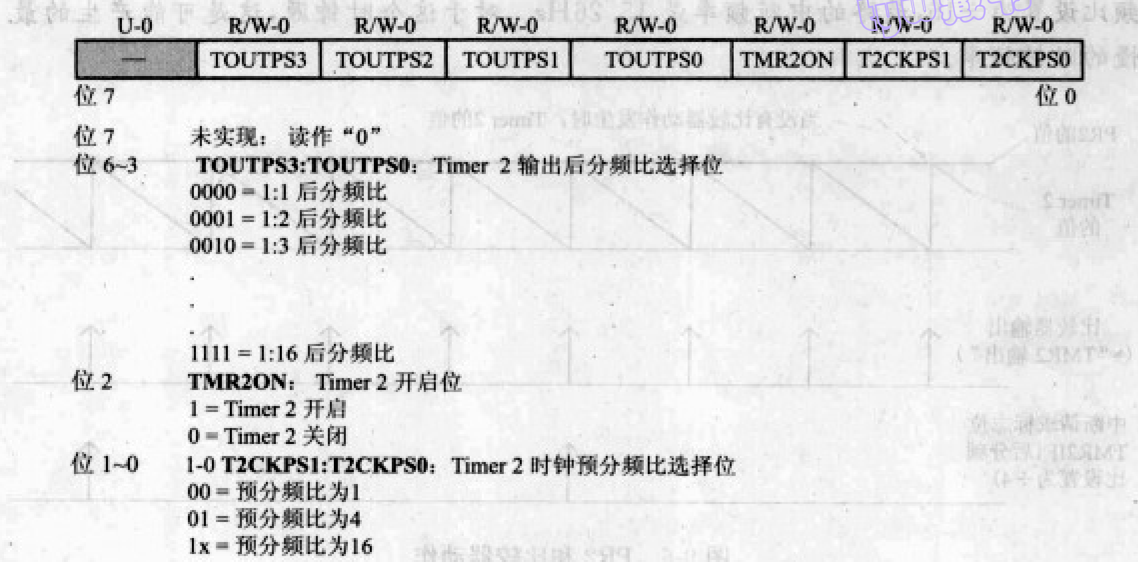


图 9-4 16F87XA 的 Timer 2 的结构框图（阴影框中所附标签为作者所加）



U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS1	T2CKPS0
位 7							位 0

位 7　未实现：读作“0”

位 6~3　**TOUTPS3:TOUTPS0**：Timer 2 输出后分频比选择位

0000 = 1:1 后分频比

0001 = 1:2 后分频比

0010 = 1:3 后分频比

.

.

.

1111 = 1:16 后分频比

位 2　**TMR2ON**：Timer 2 开启位

1 = Timer 2 开启

0 = Timer 2 关闭

位 1~0　1-0 **T2CKPS1:T2CKPS0**：Timer 2 时钟预分频比选择位

00 = 预分频比为1

01 = 预分频比为4

1x = 预分频比为16

图 9-5　Timer 2 的控制寄存器 T2CON

尽管它的结构看似很简单，但是如果这里描述的附加部件发挥出它们的作用，那么定时器将变成一个功能强大的模块。在介绍了捕捉和比较寄存器以及它所具备的脉宽调制(PWM)功能之后，定时器的用处将进一步得到扩展。

9.3.2　PR2 寄存器、比较器和后分频比器

Timer 2 有一个周期寄存器 **PR2**，存储器地址单元为 92_H(如图 7-6 所示)，它的值可以由程序员预先设置。当定时器开始运行时，比较器连续不断地将定时器的计数值与 **PR2** 的值进行比较。当计数值与 **PR2** 的值相同时，它的值在下一个周期被清零。

图 9-6 说明了这个过程。尽管可以在程序中修改 **PR2**，但是图中 **PR2** 的值是固定的。在这里，Timer 2 的计数值简化成锯齿波形，但是必须认识到波形仍然具备潜在的步进特性(如图 9-3 所示)。它递增计数直到其值等于 **PR2** 的值，这时，定时器产生复位动作并且计数值被清零。由于在检测到 2 个值相等之后的下一个周期，复位动作才会发生，所以在 2 次复位之间有(**PR2**+1)个周期间隔。图 9-4 显示这个复位动作将使定时器输出“TMR2 output”信号。这个信号可以选择用于同步串行端口(synchronous serial port，SSP)的波特率产生器，这在后续部分讲述。

设计实例 9-2

如果 AGV 中振荡器的频率为 4MHz，那么 Timer 2 可以产生的最慢的中断频率是多少？

为了获得最慢的中断频率，将 **PR2** 寄存器、预分频比、后分频比都设置为最大。如果预分频比设置为÷16，那么驱动 Timer 2 的时钟频率为 1MHz÷16，即 62.5kHz。如果 **PR2** 设置为 255，那么 Timer 2 的计数值在返回到 0 以前，它可以一直计数到最大值(即 255)。因此，定时器的复位动作频率为 62.5/256kHz，即 244.14Hz。如果将后分

频比设置为÷16，最终的中断频率是 15.26Hz。对于这个时钟源，这是可能产生的最慢的时钟频率。

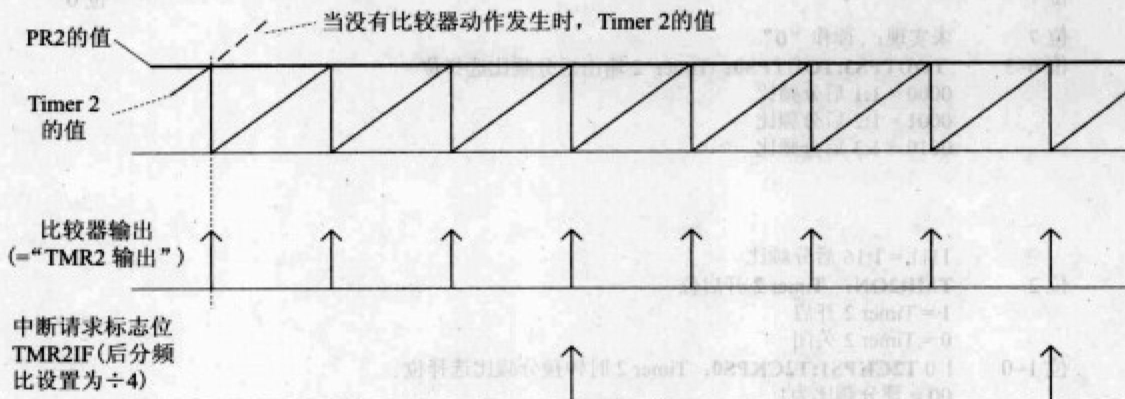


图 9-6 PR2 和比较器动作

上面描述的连续复位动作也是后分频器的输入，后分频器的输出可以作为一个中断源来使用。**T2CON** 控制寄存器的 **TOUTPSX** 位用于控制后分频比。注意，后分频器的分频比可以设置为 1～16 之间的任何值——而不仅仅局限于 2 的幂次。图 9-6 显示了当后分频比设置为 1∶4 时的情况。后分频器的输出是一个中断流，它的频率可以在一个较大的范围内进行调整。

9.4 捕捉/比较/PWM(CCP)模块

9.4.1 捕捉/比较/PWM 概论

在第 6 章的开始处，我们就提到嵌入式系统需要定时功能，它在某种程度上与我们日常生活中遇到的各种各样的时间需求是一样的。嵌入式系统确实需要一些与日常生活中等价的时间功能，例如设定一个警报或者记录事件发生的时间。这些以及其他一些定时需求可以通过在基本的计数器/定时器结构中增加一个或者多个寄存器的方法来解决。能够记录事件发生时间的寄存器称为"捕捉"寄存器。通过预先设定一个值，然后周期性地将该值与正在运行的定时器的值进行比较(与已经看到的 **PR2** 寄存器一样)，这样就设计出了一个警报器。当 2 个数相等的时候，警报被拉响。这种寄存器称为"比较"寄存器。PIC 16 系列的微控制器将这些不同的功能(还有更多未提到的功能)集成到捕捉/比较/PWM(CCP)模块中。

CCP 模块功能很强大，它工作时需要与 Timer 1 和 Timer 2 都进行交互。16F873A 微控制器有 2 个这种模块，理解它们非常重要。每个模块都有 2 个 8 位的寄存器 **CCPRxL** 和 **CCPRxH**(其中 x 取 0 或 1)。它们连在一起构成一个 16 位的寄存器，用于捕捉、比较或者形成 PWM 串的占空比。每个 CCP 模块由各自的 **CCPxCON** 寄存器控

制(图 9-7 所示)。寄存器的最低 4 个有效位确定了 CCP 模块的工作模式。从图中可以看到,CCP 模块可以被关闭,也可以设定为 PWM 模式,还可以设置为捕捉或比较模式的 4 种配置之一。我们将依次研究这些工作模式。

U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	CCPxX	CCPxY	CCPxM3	CCPxM2	CCPxM1	CCPxM0
位 7							位 0

位 7~6 未实现:读作“0”

位 5~4 **CCPxX:CCPxY**:PWM占空比的位 1和位 0
捕捉模式:
未用
比较模式:
未用
PWM模式:
它们是PWM占空比的2个LSB位。占空比的高8位在CCPRxL中

位 3~0 **CCPxM3:CCPxM0**:CCPx模式选择位
0000 = 捕捉/比较/PWM关闭(即复位CCPx模块)
0100 = 捕捉模式, 每个下降沿发生
0101 = 捕捉模式, 每个上升沿发生
0110 = 捕捉模式, 每4个上升沿发生
0111 = 捕捉模式, 每16个上升沿发生
1000=比较模式, CCP引脚初始为低电平, 比较相符时, 强制CCP引脚为高电平(CCPxIF置位)
1001=比较模式, CCP引脚初始为高电平, 比较相符时, 强制CCP引脚为低电平(CCPxIF置位)
1010 = 比较模式, 比较相符时, 产生软件中断(CCPxIF置位,CCP引脚不受影响)
1011 = 比较模式, 比较相符时, 产生特殊触发事件(CCPxIF 置位, CCPx引脚不受影响)
CCP1复位TMR1; CCP2 复位TMR1并且开始模数转换(如果启用模数转换模块)
11xx = PWM模式

图 9-7 CCP1CON/CCP2CON 寄存器(存储器地址分别为 17_H 和 $1D_H$)

9.4.2 捕捉模式

捕捉寄存器的工作有点类似于一个秒表。它们都是在事件发生时记录下事件发生的时刻。记录完毕后,在秒表中计时停止。但是,在捕捉寄存器中,时钟将继续运行,但是事件发生的时刻被记录下来。

图 9-8 显示的是 CCP1 配置成捕捉模式时的结构框图。CCP2 处于捕捉模式时,它的结构与 CCP1 相同,只是与端口 C 的位 1 共用输入。由于 CCP2 的输入引脚与端口 C 共用,因此要通过 **TRISC** 寄存器将引脚设置为输入。表示“事件”含义的外部信号连接到微控制器的 RC2/CCP1 引脚。捕捉模式的作用是当事件发生时,将 Timer 1 的计数值记录在 **CCPR1L** 和 **CCPR1H** 寄存器中。CCP 模块具有更多的灵活性,如外部信号可以设置成÷4 或者÷16 的预分频比,并且还可以设置是上升沿还是下降沿检测。图 9-7 说明了它可能的设置。除了会引发一个捕捉动作外,事件的发生也会触发一个中断,从而导致图 7-10 中断结构图中的中断请求标志位 **CCP1IF** 被置 1。CCP2 模块也有一个等价的中断请求标志位 **CCP2IF**。

235~236

9.4.3 比较模式

图 9-9 为 CCP1 配置成比较模式时的结构框图。图中所示的硬件中嵌入了一个数字比较器，它连续地将 Timer 1 的计数值与由 **CCPR1H** 和 **CCPR1L** 组成的 16 位寄存器进行比较。此时，Timer 1 必须配置成定时器或者同步计数器模式。CCP1 模块连接到外部引脚，因此引脚必须由 **TRISC** 位设置为输出模式。当比较相等时，将会触发许多事件中的一种，触发何种事件取决于控制寄存器的配置（如图 9-7 所示）。相关的输出位可以被置 1 或者清零（这两种情况下都将设置中断请求标志位）。它允许外部事件被启动或者结束。另外，在不影响输出的情况下，可以置中断请求标志位。最后，CCP1 模块还可以触发一个"特殊的事件"。对于 2 个比较器模块来说，这包括清零 Timer 1 但是不改变输出引脚。如果启用模数转换器，CCP2 模块也能启动一个模数转换操作。

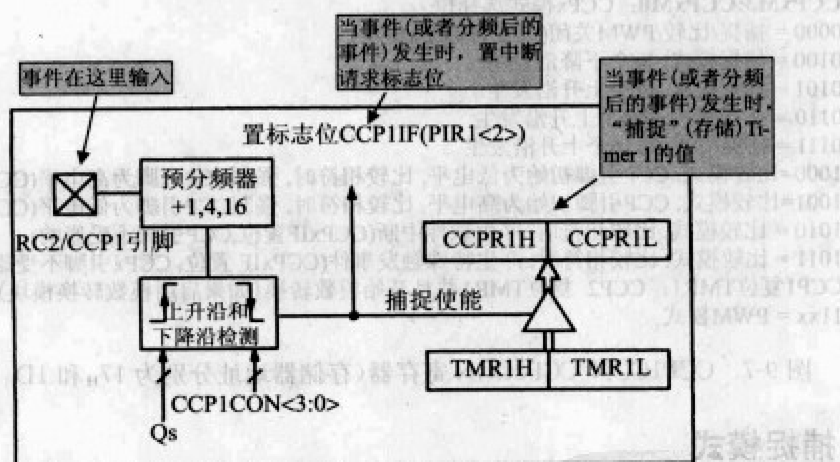


图 9-8 捕捉模式的结构框图 CCP1（阴影框中所附标签为作者所加）

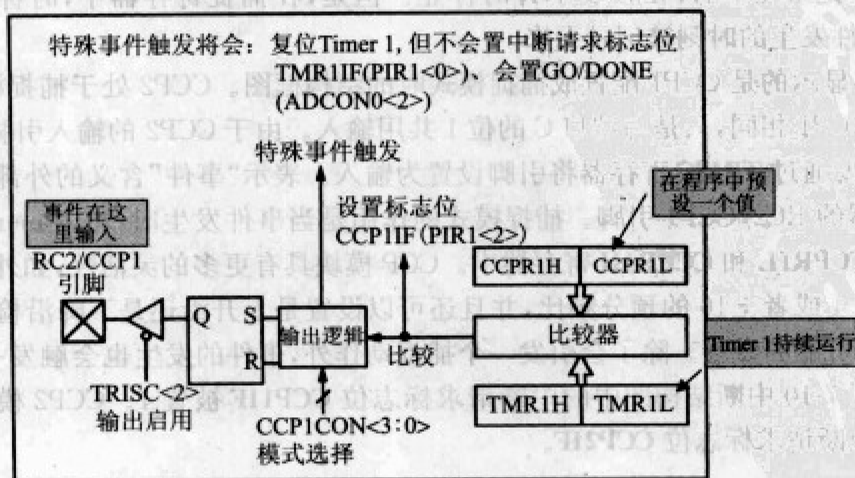


图 9-9 比较模式的结构框图(CCP1)（阴影框中所附标签为作者所加）

9.5 脉宽调制

脉宽调制(Pulse Width Modulation,PWM)是一个功能很强大的技术,它允许纯粹的数字输出控制模拟变量,且只需要单个的数据连接线。

9.5.1 PWM 的原理

对 PWM 典型应用的理解需要有一些电子学方面的知识为基础。下面我们参照图 9-10中的电路来简要回顾一下这些知识。

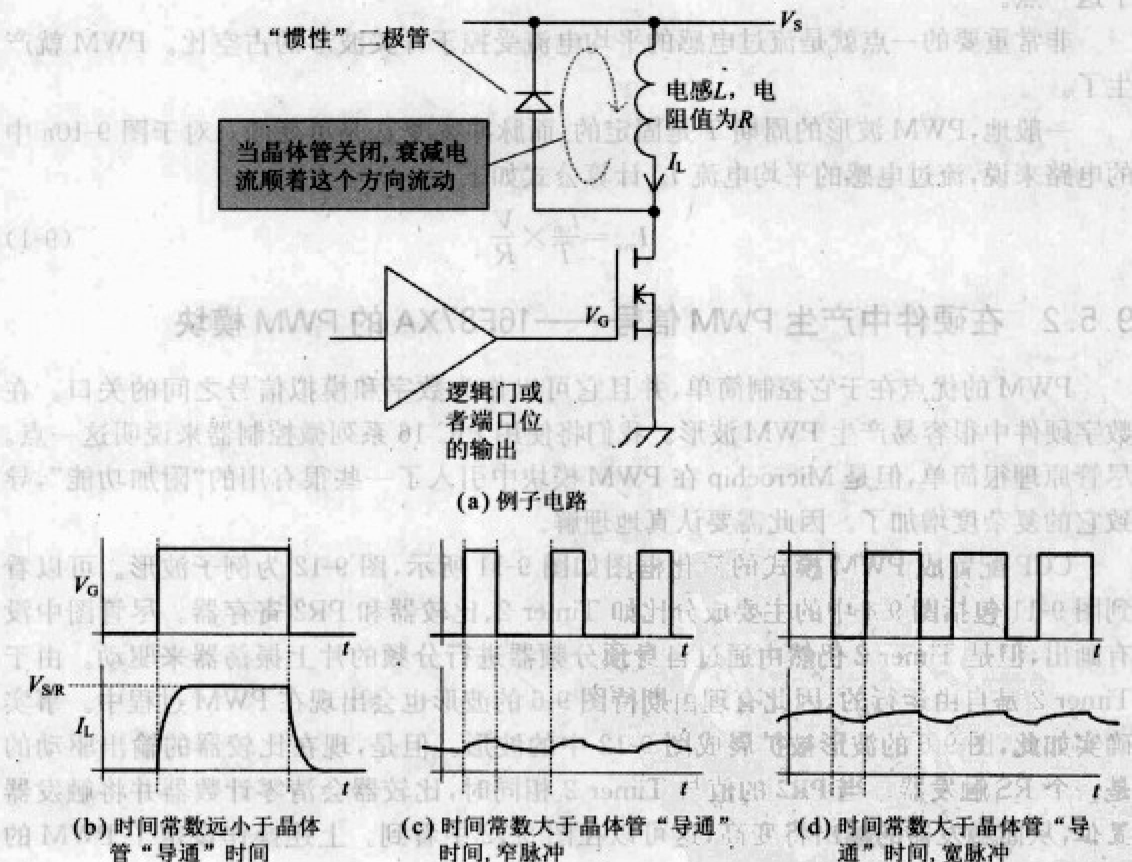


图 9-10 PWM 的原理

电感的电压—电流关系如下:

$$V = L di/dt$$

其中,V 是电感两端的电压,i 是流过电感的电流,L 是电感值。从上式可以发现,如果电压逐步改变,电流将以指数形式增加并到达一个稳定值 V/R(其中 R 是电感的电阻)。上升的速度取决于电路的常数,称为时间常数(time constant)。对于电感来说,时间常数为 L/R。当电压逐步改变时,可以计算出电流从稳定值的10%上升到90%的

时间为 2.2L/R。如果电压移除，电流下降的时间同上升的时间是一样的(假设存在一个电路用于电感电流的流入)。

238

让我们设想一下一个电感负载被晶体管开关打开和关闭，如图 9-10 所示。当晶体管关闭，衰减电流将流向惯性二极管。如果晶体管导通的时间远大于电路的时间常数，那么电流上升和下降的时间占总时间的比例很小，电流将出现图 9-10b 所示的波形。但是，如果晶体管开和关重复进行且开关周期小于电感的时间常数，那么电流将没有机会到达一个稳定的状态。当晶体管导通，电流会上升一些；当晶体管关断，电流会下降一点(电流将流过惯性二极管)。如果开和关动作持续进行，电流将上升至一个平均值，这个平均值取决定于开关波形的脉冲信号荷周比。图 9-10c 和图 9-10d 说明了这一点。

非常重要的一点就是流过电感的平均电流受控于开关波形的占空比。PWM 就产生了。

一般地，PWM 波形的周期 T 是固定的，而脉冲宽度 t_{on} 是可变的。对于图 9-10a 中的电路来说，流过电感的平均电流 I_{ave} 计算公式如下：

$$I_{ave}=\frac{t_{on}}{T}\times\frac{V}{R} \tag{9-1}$$

9.5.2 在硬件中产生 PWM 信号——16F87XA 的 PWM 模块

PWM 的优点在于它控制简单，并且它可以作为数字和模拟信号之间的关口。在数字硬件中很容易产生 PWM 波形。我们将使用 PIC 16 系列微控制器来说明这一点。尽管原理很简单，但是 Microchip 在 PWM 模块中引入了一些很有用的“附加功能”，导致它的复杂度增加了。因此需要认真地理解。

CCP 配置成 PWM 模式的简化框图如图 9-11 所示，图 9-12 为例子波形。可以看到图 9-11 包括图 9-4 中的主要成分比如 Timer 2、比较器和 PR2 寄存器。尽管图中没有画出，但是 Timer 2 仍然由通过自身预分频器进行分频的片上振荡器来驱动。由于 Timer 2 是自由运行的，因此有理由期待图 9-6 的波形也会出现在 PWM 过程中。事实确实如此，图 9-6 的波形被扩展成图 9-12 中的波形。但是，现在比较器的输出驱动的是一个 RS 触发器。当 **PR2** 的值与 Timer 2 相同时，比较器会清零计数器并将触发器置位，从而触发器的输出将变高，这可以在图 9-12 中看到。上述操作设定了 PWM 的周期。

建立了 PWM 的周期之后，让我们继续考虑如何确定脉冲宽度。引入了第 2 个比较寄存器来做这件事。它由 **CCPR1H** 寄存器和第 2 个比较器组成。如电路逻辑图所示，比较器比较 TMR2 和 **CCPR1H** 寄存器，一旦它们相同将复位输出触发器，并将输出变为 0。这个比较器确定了脉冲宽度，这也可以在图 9-12 中看到。当需要改变脉冲宽度时，程序员可以修改 **CCPR1L** 寄存器，它作为 **CCPR1H** 寄存器的缓冲。只有当一个 PWM 周期完成之时，**CCPR1L** 寄存器的值才会传递到 **CCPR1H** 寄存器，这样是为了避免在修改脉宽的过程中引入输出错误。

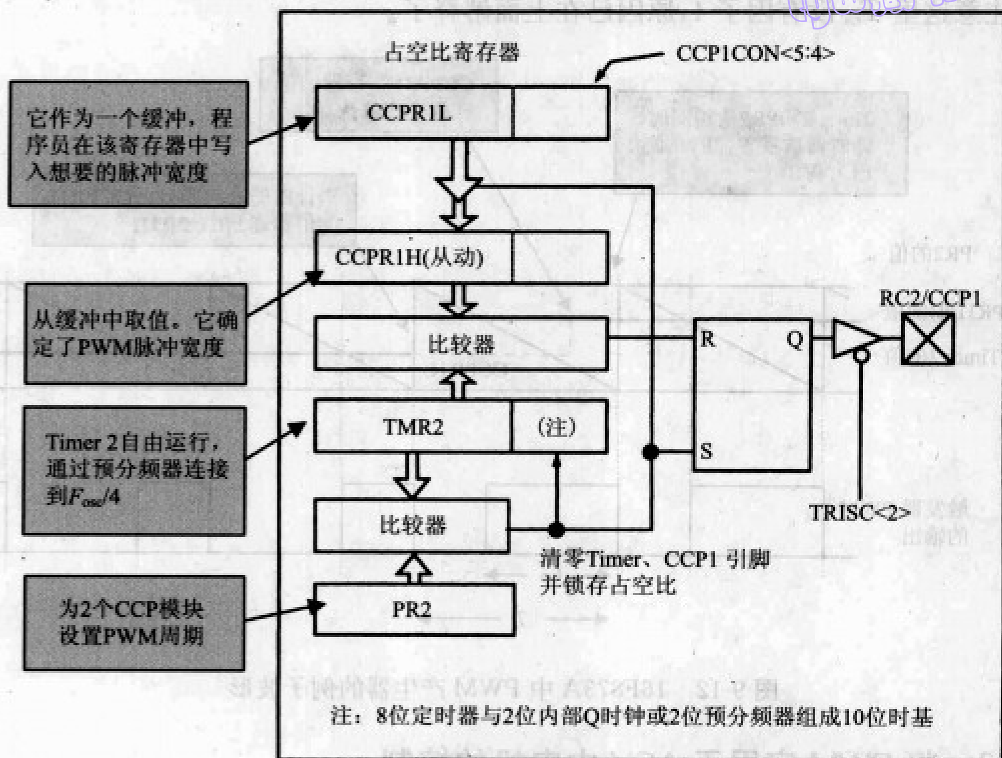


图 9-11　简化的 PWM 模块框图(CCP1 的,CCP2 结构类似)(阴影框中所附标签为作者所加)

图 9-11 的框图比图 9-4 更加复杂,由于图中 3 个寄存器由 8 位"扩展"到 10 位。这增加了 PWM 的分辨率。**CCPR1L** 使用了 **CCP1CON** 寄存器的 2 位(这可以在图 9-7 中看到)。内部 2 个锁存器对 CCPR1H 进行了扩展,图 9-11 的注 1 对 Timer 2 的扩展进行了描述。由于扩展了 2 位,10 位版本的定时器可以直接由内部振荡器信号驱动,而不再用分频之后的。如果使用了预分频器,定时器的驱动频率就不是通常的 $F_{osc}/4$。但是,需要注意的是 **PR2** 寄存器继续保持 8 位的宽度。这意味着 PWM 周期的分辨率仍然只有 8 位。

239~240

比较 **PR2** 寄存器与 8 位 Timer 2 计数值确定了 PWM 的周期 T。它的值计算如下:

$$T = (\mathbf{PR2}+1)\times(\text{Timer 2 的输入时钟周期})$$
$$= (\mathbf{PR2}+1)\times[T_{osc}\times 4\times(\text{Timer 2 的预分频值})] \quad (9\text{-}2)$$

比较扩展的 **CCPR1H** 寄存器(10 位)与扩展的 Timer 2(10 位)确定了 PWM 的脉冲宽度 t_{on}。它的值计算如下:

$$t_{on} = (\text{脉冲宽度寄存器})\times(\text{PWM 模块的输入时钟周期})$$

其中"PWM 模块的输入时钟周期"是输入到扩展的 Timer 2 的输入时钟,"脉冲宽度寄存器"是 **CCPR1H** 寄存器中的数值。因此,

$$t_{on} = (\text{脉冲宽度寄存器})\times[T_{osc}\times(\text{Timer 2 预分频值})] \quad (9\text{-}3)$$

注意这里 T_{osc} 没有因子 4，原因已在上面解释了。

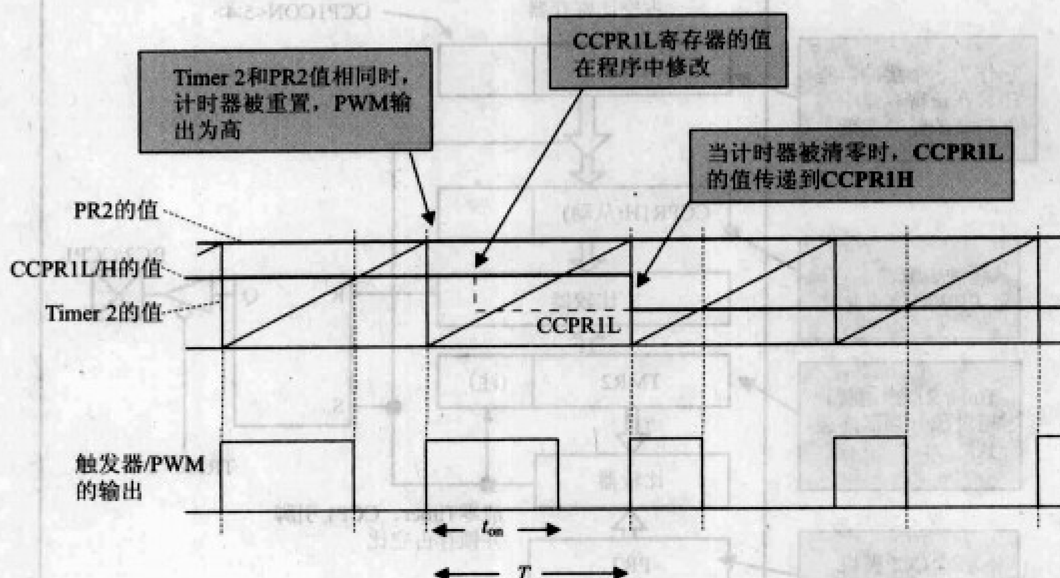


图 9-12　16F873A 中 PWM 产生器的例子波形

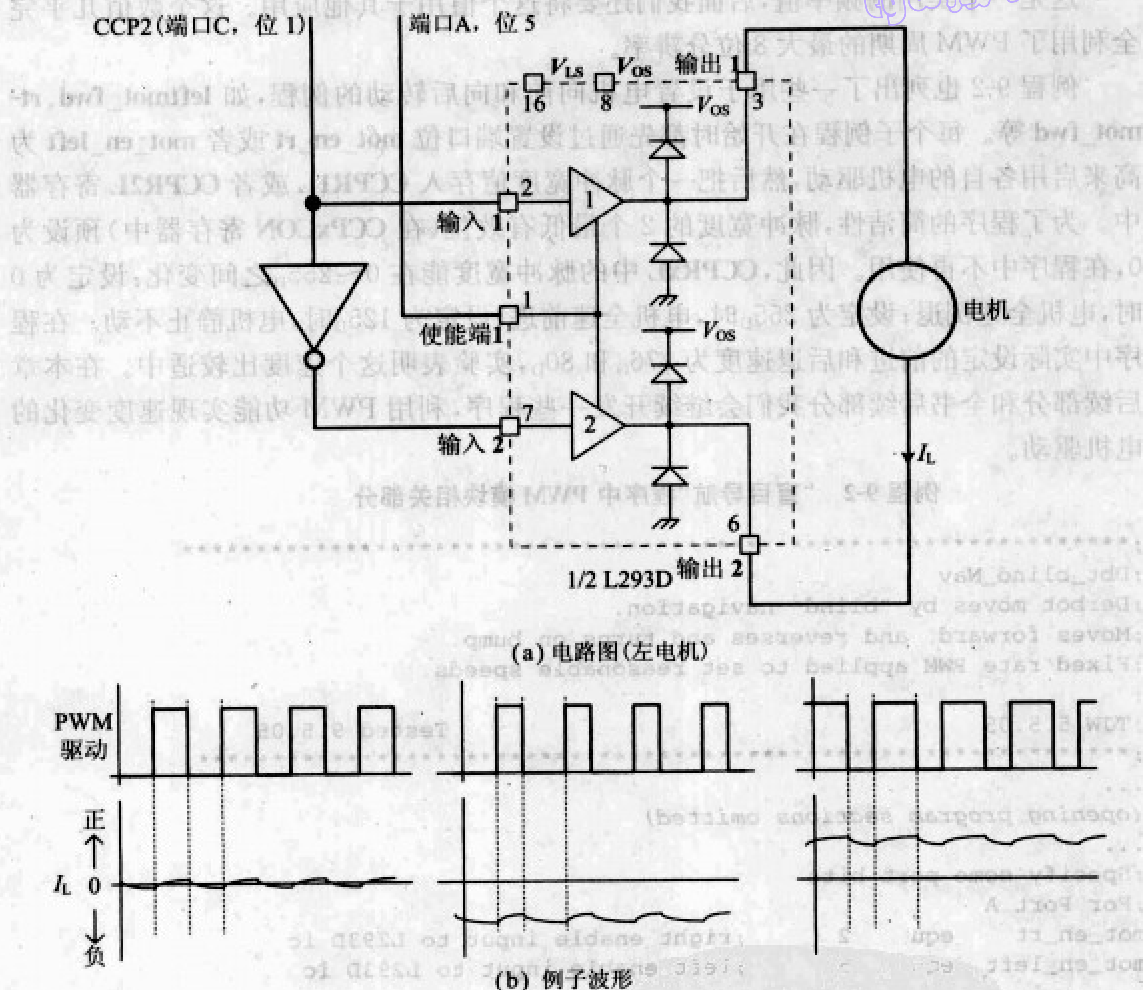
9.5.3　将 PWM 应用于 AGV 中电机的控制

现在我们来看一下如何在 AGV 中应用 16F873A 的 PWM 模块。首先，我们需要理解 PWM 模块是如何应用到电路设计中的。在第 8 章我们看到 H-桥被用于电机的双向电压驱动。如果将 PWM 而不是开/关驱动应用于 H-桥，那么就可以实现一个持续变化的双向驱动。

AGV 中左电机的电路图如图 9-13a 所示。L913D(见图 8-27)集成电路的一半被用于形成一个 H-桥，它的输出驱动电机。H-桥的 2 个输入之间连接了一个反相器，因此当任何一个输入的电平为逻辑 1 时，另外一个输入的电平就为逻辑 0，反之亦然。如果标记为"桥驱动"的输入电平为 1，电机将向一个方向旋转；当电平为 0 时，将向相反的方向旋转。

如果现在将 PWM 连接到 H-桥的驱动，有趣的事情就会发生。如果 PWM 输出的是一个频率足够高的正方形波，电机将静止不动，这是因为平均电流为 0，图 9-13b 说明了这一点。如果桥驱动的波形的占空比小于 1，那么产生的电流是一个负的平均值。如果 H-桥驱动的波形的占空比大于 1，那么产生的电流是一个正的平均值，这也可以从图 9-13b 中看到。如果在整个可变范围内(从绝对最小到绝对最大)调节 PWM 脉宽，电机的速度将在正反方向上持续变化。

在 AGV 的全电路图 A3-1 中，可以发现 CCP1 驱动左电机，CCP2 驱动右电机。使能信号线连到端口 A 的位 2 和位 5。



(a) 电路图(左电机)

(b) 例子波形

图 9-13 通过 H-桥将 PWM 应用于 AGV——实现持续变化的双向控制

第 8 章例程 8-4 的“盲目导航”程序是一个最简单的程序,它能够让我们实际研究一下在 AGV 中如何使用 PWM 模块。程序中与 PWM 相关的部分在前一章省略,现在出现在例程 9-2 中。

程序开始处是 PWM 的初始化。通过写 **T2CON** 寄存器,Timer 2 被打开,并且将驱动时钟配置成不带预分频和后分频的模式(见图 9-4)。通过修改 **CCP1CON** 和 **CCP2CON** 寄存器,将 2 个 CCP 模块设置为 PWM 模式。最后将十进制数 249_D 存入 **PR2** 寄存器。注意到时钟振荡器的频率为 4MHz,利用等式(9-2)可计算出 PWM 的周期:

$$T=(\mathbf{PR2}+1)\times[T_{osc}\times4\times(\text{Timer 2 预分频值})]$$
$$=250\times(250\text{ns}\times4\times1)$$
$$=250\mu\text{s}$$

PWM 频率=4.00kHz

242

这是一个实用的频率值，后面我们还要将这个值用于其他应用。这个数值几乎完全利用了 PWM 周期的最大 8 位分辨率。

例程 9-2 也列出了一些用于设置电机向前和向后转动的例程，如 **leftmot_fwd**、**rtmot_fwd** 等。每个子例程在开始时都先通过设置端口位 **mot_en_rt** 或者 **mot_en_left** 为高来启用各自的电机驱动，然后把一个脉冲宽度值存入 **CCPR1L** 或者 **CCPR2L** 寄存器中。为了程序的简洁性，脉冲宽度的 2 个最低有效位(在 **CCPxCON** 寄存器中)预设为 0，在程序中不再使用。因此，**CCPRxL** 中的脉冲宽度能在 0～255_D之间变化：设定为 0 时，电机全速倒退；设定为 255_D时，电机全速前进；设定为 125_D时，电机静止不动。在程序中实际设定的前进和后退速度为 176_D和 80_D，实验表明这个速度比较适中。在本章后续部分和全书后续部分我们会继续开发一些程序，利用 PWM 功能实现速度变化的电机驱动。

例程 9-2 “盲目导航”程序中 PWM 模块相关部分

```
;**************************************************************
;Dbt_blind_Nav
;Derbot moves by "blind" navigation.
;Moves forward, and reverses and turns on bump.
;Fixed rate PWM applied to set reasonable speeds.
;
;TJW 5.5.05                                  Tested 9.5.05
;**************************************************************
...
(opening program sections omitted)
...
;Specify some port bits
;For Port A
mot_en_rt     equ    2      ;right enable input to L293D ic
mot_en_left   equ    5      ;left enable input to L293D ic
...
     bcf      status,rp0             ;select bank 0
;set up PWM
     movlw    B'00000100'   ;switch on Timer2, no pre or postscale
     movwf    t2con
     movlw    B'00001100' ;enable PWM
     movwf    ccp1con
     movwf    ccp2con
     movlw    0f9
     movwf    pr2
...
(main program omitted - appears as Program Example 8.4)
...
;motor drive subroutines
leftmot_fwd                  ;sets left motor running forward
     bsf      porta,mot_en_left
     movlw    D'176'
     movwf    CCPR2L
     return
```

243

```
rtmot_fwd bsf porta,mot_en_rt
      movlw  D'176'
      movwf  CCPR1L
      return

leftmot_rev bsf porta,mot_en_left
      movlw  D'80'
      movwf  CCPR2L
      return

rtmot_rev bsf porta,mot_en_rt
      movlw  D'80'
      movwf  CCPR1L
      return
```

9.6 软件产生 PWM

尽管硬件产生 PWM 是一种通用的方法并且使用很简单，但是实际操作中并不总是可以获得硬件资源。例如在一个注重成本的应用中可能选择了一个不带 PWM 模块的微控制器。另外，硬件 PWM 模块的数量可能不足以满足应用中的需求。PWM 是一个功能既可用硬件也可用软件方式实现的典型模块。如果使用软件方式实现 PWM，那么唯一需要利用的硬件资源只是一个简单的设置为输出的端口引脚。

可以只基于软件延时循环的方法来产生 PWM 波形，这在第 5 章已经讲述过。参考文献 1.1 中的图 5-4 给出了软件延时方法的一个可能的流程。但是采用这种方法时，延时程序几乎完全占用了 CPU 资源，因此在实际应用中受到限制。

另外一种纯软件编程产生 PWM 波形的方法是使用定时器中断来设置周期。这种方法可以简化编程的复杂性，并且允许 CPU 可以做其他的事情。需要做的工作是配置定时器以适当的频率向 CPU 发出溢出中断，这在本章的 9.2.4 节已讲到。CPU 每接收到一个中断时，将 PWM 的输出设置为高，然后调用一个软件延时程序。延时结束后将 PWM 的输出变低，如图 9-14 所示。

244

9.6.1 一个软件产生 PWM 的例子

例程 9-3 是解释软件产生 PWM 波形的一个很好的例子。程序通过驱动 AGV 中的伺服装置来旋转超声波检测器。16F873A 只有 2 个 PWM 模块，并且它们都已经用于左右电机的驱动。图 8-23 是驱动伺服装置需要的驱动脉冲波形。这种波形非常适合采用软件方式产生。相比于输出波形的周期来说，脉冲的宽度很窄，因此可以利用 CPU 来产生这个相对较短的延时。程序从参考 0°开始转动伺服装置的转轴，每次递增

45°,一直到 180°。如果要求伺服装置立即[1]改变旋转角度,电流的消耗会很大,因此程序在不同的旋转角度改变之间有一段缓冲[2]。

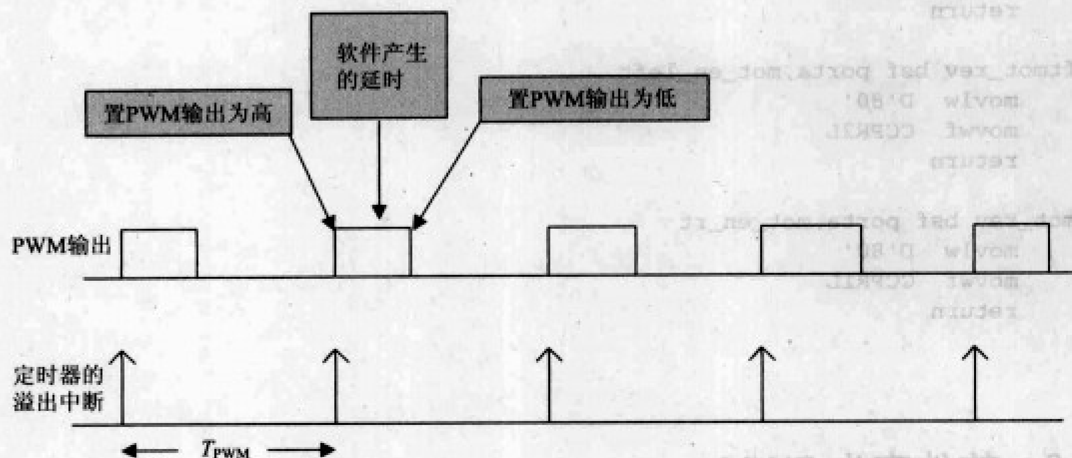


图 9-14 使用定时器中断产生 PWM

由于 Timer 0 和 Timer 1 已经用于轴角编码器,因此可能只有 Timer 2 的定时器溢出功能可用。但是,它好像已经用于 PWM 功能。这里我们来看一下定时器如何能够被用于 2 个看似不相关的功能。定时器仍然用于 PWM 功能,但是通过后分频器(见图 9-4)分频得到的溢出中断是可以用于其他新的应用的。PWM 和驱动伺服装置这两个应用之间有一些冲突:伺服装置需要一个 20ms 的周期,但是 PWM 的频率需要保持在几千赫兹的范围之内。这时,灵活的后分频器在这里体现出它的优势。正如驱动电机的 PWM 模块的设置,可以看到如果 **PR2** 设定为 249_D,那么定时器 2 的溢出中断的频率为 4kHz。当后分频设置为 ÷16 时,中断频率为 250Hz(周期为 4ms)。这个周期并不是我们想要的 20ms,但是可以很容易地在程序中做到每 5 次中断输出一个脉冲。

上面所述的设置可以在例程中看到。寄存器 **PR2** 的载入值设定为 249_D,Timer 2 的后分频设定为 ÷16。随后 3 个软件计数器的值被预先设置:**int_cntr** 对触发的中断次数进行计数;**pulse_cntr** 对同一个脉冲宽度值输出了多少个脉冲进行计数;**past_pulse** 保存前一次输出的脉冲宽度,用于确定是否需要在前一次脉冲宽度上进行旋转角度的缓冲。然后启用 Timer 2 的中断,程序开始进入等待中断的循环。所有后续的程序活动都在中断服务程序(Interrupt Service Routine, ISR)中进行。

245

在中断服务程序中,**int_cntr** 的值被减 1,然后判断是否需要输出一个脉冲。如果需要,**pulse_cntr** 的值减 1,然后判断是否需要改变输出脉冲的宽度。如果要改变脉冲

① 立即是指一次完成 45°的递增。——译者注

② 分多次完成 45°的递增。——译者注

宽度，servo_dirn 的值递增。它作为读取脉宽查找表的索引，在程序标号 ISR1 处，从查找表中读取要求的脉宽，随后存放在存储器单元 **pw_cntr** 中。**pw_cntr** 中储存的这个值实际上确定的是调用一个 20μs 延时程序的次数，从而设定 PWM 输出脉冲的宽度。

如果需要，角度递增缓冲在这里产生。将 **pw_cntr** 中存放的值与存放在存储器单元 **past_pulse** 中上一次的输出脉宽值进行比较。如果相等，直接输出脉冲。否则，通过测试进位标志位来判断哪个值较大。然后，根据判断结果，将 **pw_cntr** 的值相应地增加或者减少来重新设定脉宽。随后，使用一个简单的延时循环来输出这个脉冲。

由于角度改变是平滑递增的，因此在实际中程序运行很正常。

例程 9-3 使用软件产生 PWM

```
;************************************************************************
;Dbt_servo_tst
;Rotates servo position in 45 degree steps from 0 degs to 180 degs.
;Servo PWM drive period, is generated with interrupt on overflow
;(Timer 2), with pulse width in software.
;Ramp between each step is generated.
;TJW 26.5.05                                         Tested 30.8.05
;************************************************************************
...
(opening comments omitted)
        cblock 20
delcntr1       ;used in delay5 & delayADC SRs
delcntr2
temp           ;a temp location, to be used only in consecutive instructions
int_cntr       ;counts incoming interrupts
pulse_cntr     ;counts how many pwm pulses sent to servo, before moving to
                   ;next posn.
pw_cntr        ;counter used to set width of pulse to servo
servo_dirn     ;little counter which determines direction servo points, used
                   ;as look-up table pointer
past_pulse     ;holds value of most recent pulse width, used with demand value
                   ;to determine actual
        endc
...
        org   00
        goto  begin
;
        org   04
        goto  ISR
;Initialise
;set up SFRs in Bank 1
begin   bcf     status,rp1
        bsf     status,rp0  ;select memory bank 1
...
        movlw   B'11110001'   ;set port B bits, bit 3 is servo PWM op
        movwf   trisb
...
```

```
        movlw   D'249'              ;set Timer 2 Overflow period
        movwf   pr2
;set up SFRs in Bank 0
        bcf     status,rp0          ;select bank 0
        movlw   B'01111100'  ;switch on Timer2, no prescale, /16 postscale
                                ;giving 4ms interrupt prd (with pr2=250)
        movwf   t2con
;
;Initialise counter values
        movlw   D'5'  ;preset interrupt counter
        movwf   int_cntr
        movlw   D'200'
        movwf   pulse_cntr    ;preload pulse counter
        movlw   D'60'  ;start with intermediate pulse width
        movwf   past_pulse
...
(opening program section omitted)
...
;Enable interrupts
        bcf     pir1,tmr2if         ;clear pending interrupts
        bsf     status,rp0          ;select memory bank 1
        bsf     pie1,tmr2ie         ;enable Timer 2 interrupt
        bcf     status,rp0          ;select memory bank 0
        bsf     intcon,peie         ;enable peripheral interrupts
        bsf     intcon,gie
wait goto wait         ;let ISR do the work
;
;*******************************************************************
;ISR is here. Interrupts occur every 4ms. Count 5, and on 5th
;emit pulse to Servo. Pulse length will depend on servo_dirn setting
;*******************************************************************
ISR     decfsz int_cntr      ;decrement interrupt counter. Action occurs
                                ;only if it is 0.
        goto    intend
        decfsz pulse_cntr    ;here if a pulse to be output, test if pulse
                                ;width is to change
        goto    ISR1
        incf    servo_dirn,1 ;here if pulse width changing, point to new
                                ;duration
        movlw   D'200'       ;reload pulse counter, 200 pulses will be emitted
        movwf   pulse_cntr      ;at new pulse width
;now determine pulse width, calculating ramp value if needed
ISR1    movf    servo_dirn,0 ;get demand pulse duration from Table
        andlw   07           ;use only 3 lsbs of servo_dirn
        call    int_table
        movwf   pw_cntr      ;this is demand value, may need to ramp towards it
;now determine ramp value, if needed
        subwf   past_pulse,0 ;compare demand value with most recent pulse
        btfsc   status,z
        goto    pulse_op     ;if equal, go direct to pulse
        btfsc   status,c     ;if carry clear, demand>past
        goto    $+3
```

```
       incf    past_pulse,1 ;here if demand>past, hence ramping up
       goto    $+2
       decf    past_pulse,1 ;here if demand<past, hence ramping down
       movf    past_pulse,0 ;and save new value for next time round
       movwf   pw_cntr
;now send pulse
pulse_op bsf   portb,3      ;set pulse high
pw_loop call   delay20u
       decfsz pw_cntr
       goto    pw_loop
       bcf     portb,3       ;clear pulse
       movlw   D'5'   ;reload interrupt counter
       movwf   int_cntr
intend bcf pir1,tmr2if ;clear interrupt flag
       retfie
;Table for servo positions
int_table addwf pcl
       retlw D'20'              ;400us delay, 0 degrees
       retlw D'40'              ;800us delay, 45 degrees
       retlw D'60'              ;1200us delay, 90 degrees
       retlw D'80'              ;1600us delay, 135 degrees
       retlw D'100'             ;2000us delay, 180 degrees
       retlw D'80'              ;1600us delay, 135 degrees
       retlw D'60'              ;1200us delay, 90 degrees
       retlw D'40'              ;800us delay, 45 degrees
;
;*****************************************************************
;SUBROUTINES
;*****************************************************************
;introduces delay of 20us
delay20u  movlw 5    ;5 cycles called, each taking 3us,
          ;plus call, return (2 ea), and 2 move insts
;                    less one cycle lost when last goto is hopped
       movwf         delcntr1
dela   decfsz        delcntr1,1 ;3 inst cycles in this loop, ie 3us
       goto dela
       return
;
...
(other delay subroutines omitted)
...
       end
```

9.6.2 与存储器定义和跳转相关的汇编伪指令

注意到在上面的程序列表中我们使用了2个新的汇编伪指令。在程序最开始处，一对汇编伪指令 **cblock** 和 **endc** 是用来定义数据存储器类型的变量块。这种变量定义方式取代了一长串 **equ** 语句的定义方式。在中断服务程序中，跳转通过下面的语句来实现：

248

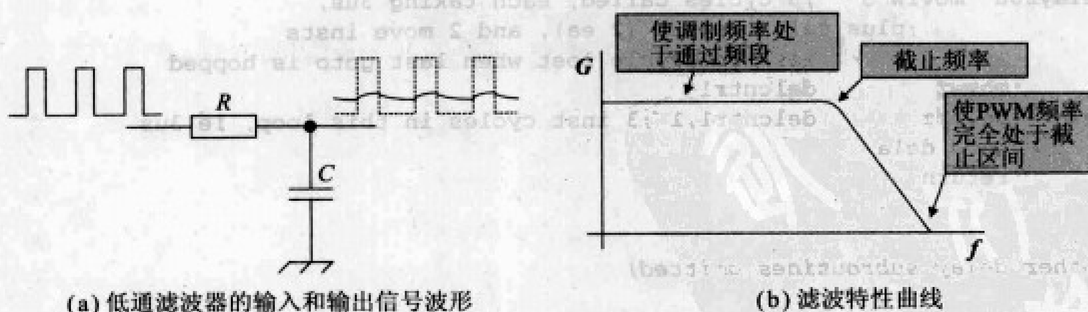
```
btfsc   status,c     ;if carry clear, demand>past
goto    $+3
incf    past_pulse,1  ;here if demand>past, hence ramping up
goto    $+2
```

在这里,符号$代表当前程序计数器值。因此,**goto $+3** 的作用是强制程序向前跳3行。也可以使用负值,那么程序将回跳。这个方法对于局部跳转中一些行的前跳或者回跳非常有用,它不使用行标号即可实现跳转。但是对于远距离的跳转,这种方法就不实用了。因为对 **goto** 指令到其跳转的目标行之间的程序修改会改变跳转距离,导致原来设定的跳转行数无效。

9.7 使用 PWM 进行数模转换

尽管 PWM 模块可能主要用于负载控制,但是它也可以作为一个简单但很有效的数模转换器(DAC)。如果 PWM 波形的占空比固定,并且通过一个具有合适截止频率的低通滤波器,那么数字流将变为一个具有残留波形的直流电压。如果调制 PWM 输出波形的占空比,就可以产生一个可变的输出电压。

图 9-15 以一个简单的方式说明了这个方法。PWM 波形经过一个 RC 滤波器,这个滤波器具有平滑作用,如图所示。滤波器的特性如图 9-15b 所示,这样的特性使得 PWM 的频率正好处于截止频率区间。如果调制频率处于滤波器的通过频率段,那么 PWM 频率被有效地阻塞,只有调制频率可以通过。这种特性是非常吸引人的。



(a) 低通滤波器的输入和输出信号波形　　(b) 滤波特性曲线

图 9-15　对 PWM 进行滤波以产生模拟电压

一个使用 PWM 进行数模转换的例子

249

在 AGV 电路中有 2 个低通滤波器部件,它们的输出分别是 TP1 和 TP2(见图 A3-1)。它们在 AGV 电路图中出现只是为了说明如何使用 PWM 进行数模转换,而不用于操作 AGV。如果在 AGV 中使用 PWM 进行数模转换的实验,应该置电机的使能位为 0。

本书附属资源中的程序 **dbt_pwm_qrtr_sinwave** 是一个简单地说明这个技术的

例子。例程 9-4 为它的部分程序代码。程序使用全 10 位的分辨率来设置 PWM 的时间周期。它通过依次从一个 **Sin_Table** 查找表中取样来产生 1/4sin 波形。存储器单元 **pointer** 用来指出当前程序要访问的查找表索引。查找表共有 45 个采样点，每个点都是 2 个字节的数。它们依次是从 0°、2°、4°，一直到 90°的 sin 函数值，这些值都是比例可调的二进制值。但是在这个程序中只有低字节的最高 2 个有效位被使用。

在程序中，一旦初始化过程完成，程序只剩下一个简单的循环，这个循环从标号 **sin_loop** 开始。采样值的高字节直接传输到 **ccpr1l** 寄存器。采样值的低字节首先被调整，然后它的 2 个最高有效位放入 **ccp1con** 的第 5 位和第 4 位。在操作的同时，不能修改 **ccp1con** 的其他位。可以按照程序清单中给出的方法来操作。最后，插入了一段延时 1ms 的程序 **delay1**。这个延时程序以及后续程序的运行时间确定了 sin 波形的频率。

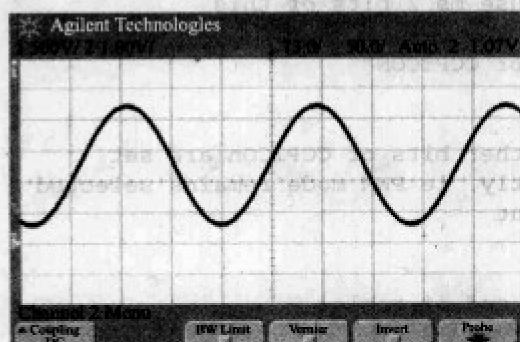
例程 9-4　使用 PWM 产生 1/4sin 波形

```
;*****************************************************************
;dbt_pwm_qrtr_sinwave
;Demonstrates quarter sin wave output, on CCP1
;Uses full ten bits of PWM period setting
;TJW 9.7.05                                     Tested 11.7.05
;*****************************************************************
...
(all opening sections of program omitted)
...
         clrf    pointer
sin_loop movf    pointer,0
         call    sin_table      ;get more significant byte of sample
         movwf   ccpr1l         ;move it to the PWM output
         incf    pointer,1      ;increment the pointer
         movf    pointer,0
         call    sin_table      ;get the less significant byte
         andlw   B'11000000'    ;we only use ms 2 bits of this
         movwf   temp
         bcf     status,c       ;adjust for CCP1CON
         rrf     temp,1
         rrf     temp,0
         iorlw   B'00001100'    ;ensure other bits of CCP1COn are set
                                  ;correctly, ie PWM mode remains selected
         movwf   ccp1con        ;and output
         incf    pointer,1
         movf    pointer,0
         sublw   D'92'          ;test for end of table
         btfsc   status,z
         clrf    pointer        ;reset pointer
         call    delay1
         goto    sin_loop
;
```

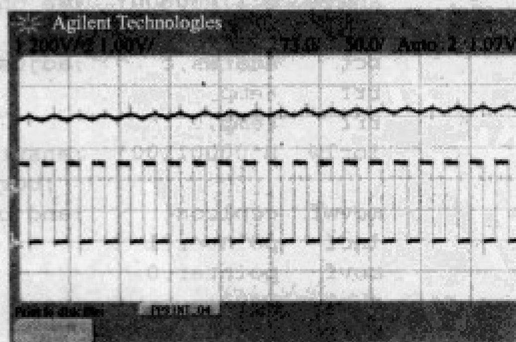
```
Sin_Table
        addwf pcl,1
        retlw 00      ;0 degrees, higher byte
        retlw 00      ;0 degrees, lower byte
        retlw 03      ;2 degrees, higher byte
        retlw 5A      ;2 degrees, lower byte
        retlw 06      ;4 degrees, higher byte
        retlw 0B2     ;4 degrees, lower byte
        ...
```

本书附属资源中还有一个程序 **dbt_pwm_full_sinwave**。它使用相同的查找表来产生整个 sin 波形，但是比 1/4sin 波形产生版本稍微复杂一些。它将所有的采样点都加上一个中间值 125_D。这代表 sin 函数的零偏移值。在第 1 个 sin 象限，幅度逐步上升。在第 2 个象限，幅度逐步下降。在第 3 个象限，幅度再次逐步上升，但是每一个采样点减去 125_D。[①] 在第 4 个象限，幅度又逐步下降，采样点减去 125_D。它们可以在整个程序清单中看到。

dbt_pwm_full_sinwave 程序的输出波形如图 9-16a 所示，PWM 的输出经过了一个 100nF/20kΩ 的滤波器件（其截止频率大约为 80Hz）。注意图中垂直设置和水平设置。通道 2 显示了 sin 波形的 2 种情况。图 9-16a 显示了一个令人满意的 sin 波形。图 9-16b 显示了同一个 sin 波形的详图和产生这个 sin 波形的 PWM 波形。在图 9-16b 中可以看到 PWM 的周期为 250μs（即频率为 4kHz）。sin 波形是一个伴随 PWM 波形呈现锯齿形上升和下降的模拟波形，这个特征可以很明显地从信号的跟踪波形中看到，总体信号电压很明显地呈上升趋势。由于 PWM 模块的频率是 4kHz，并且在程序中使用了一个 **delay1** 延时程序，微控制器每 4 或每 5 个 PWM 周期会输出一个采样点。可以看到整个 sin 波形的周期为 190ms（50ms 的 3.8 分频）。这个时间等于拼成一个完整周期的 sin 波形的 180 个采样点时间的总和，每一个采样点的时间包括 1ms 延时和程序执行时间。



(a) 输出的sin波形



(b) 下部：PWM波形。上部：模拟波形输出详图

图 9-16 由 PWM 产生的 sin 波形

① 使产生的 sin 波形处于偏移零值以下。——译者注

使用 PWM 方法进行数模转换是简单且有效的方式。但是它也有一些缺点，在使用这种方法之前，必须了解这些缺点。这些缺点如下所示。

- ☐ 输出的模拟信号的电压直接受制于 PWM 波形的逻辑电平。而逻辑电平是否稳定取决于电源电压的准确度、接地通路是否顺畅以及器件工艺。总的来说，精确的数模转换一般不使用这种方法。
- ☐ 由于低通滤波的影响，这种方法不能用来产生快速变化的模拟信号。针对这种情况，可以采用一个截止频率较高的滤波器或者通过提高 PWM 的波形频率来解决。通过减少 **PR2** 寄存器的值可以提高 PWM 的频率。虽然提高了 PWM 频率，但是应该注意到精度却在下降。一般地，高的频率需求总是同高的分辨率需求相互冲突。
- ☐ 在输出的模拟信号上总会存在一些残留的波形。

9.8 频率测量

9.8.1 频率测量的原理

频率测量是计数和定时这 2 种功能的重要应用。从本质上说，频率测量就是在一个固定已知的周期内对某种事件发生的次数进行计数，如图 9-17 所示。频率测量的使用形式是多样的：在 Geiger 计数器中每分钟接收了多少个放射性元素；电子或者声学测量中每秒的周期数（即赫兹）；在速度测量中，单位时间内的转数。在测量时，计数器和定时器都需要使用：定时器用于产生测量需要的参考周期时间[①]；计数器用来对这个周期内对事件发生的次数进行计数。

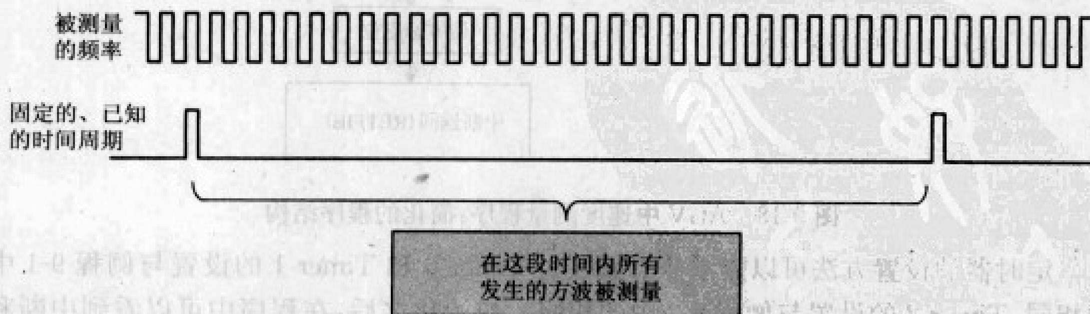


图 9-17 频率测量的原理

9.8.2 AGV 中的频率（速度）测量

在 AGV 中频率测量方法用来测量行驶的速度，本章后面一节将应用它进行速度

252

① 后面可能称其为时基。——译者注

控制。如图 9-17 所示，进行频率测量首先需要的是一个精确的时基。依据这个时基，测量才能进行下去。尽管可以使用 16F873A 中的任何一个定时器来获得这个时基，但是 Timer 0 和 Timer 1 都配置成计数模式用于车轮轴角编码器中，因此不能再加以利用。Timer 2 似乎致力于 PWM 功能，但是如同 9.6 小节所述的软件产生 PWM 波形一样，进一步的调查表明它仍然可以被利用。在 9.6 节中我们已经讲解了如何在不影响 PWM 操作的情况下产生一个 4ms 的中断周期。对于大部分的频率测量来说，这个时间太短。但是我们可以基于这个时钟周期特定次数的迭代来建立测量周期。

例程 9-5 显示的是程序 **Dbt_speed_meas** 的部分代码，它是一个简单的速度测量程序，可以在本书附属资源中找到它的源代码。程序简单地打开电机，以固定的 PWM 频率来驱动电机转动。程序的结构如简化的流程图 9-18 所示。程序中，初始化系统，启用电机和 Timer 2 中断。Timer 2 每 4ms 中断一次，在第 250 次中断（即每秒）时，读取 Timer 0 和 Timer 1 的计数累积值。

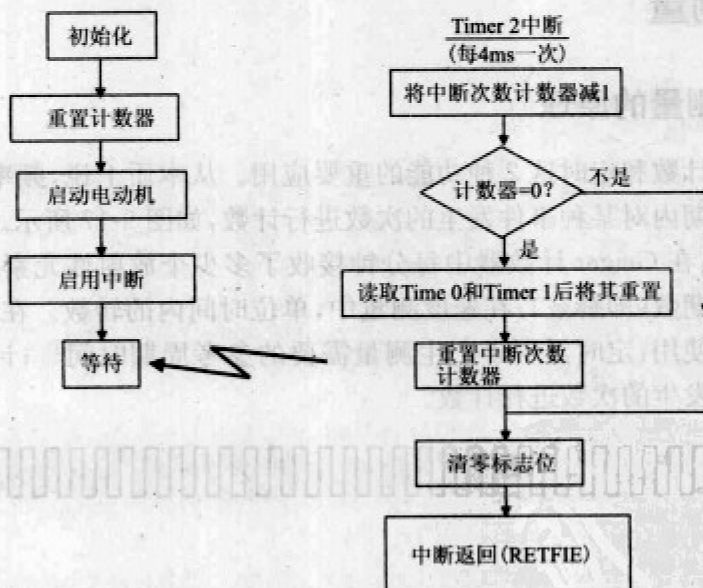


图 9-18 AGV 中速度测量程序：简化的程序结构

定时器的设置方法可以查看程序清单。Timer 0 和 Timer 1 的设置与例程 9-1 中的相同，Timer 2 的设置与例程 9-3 中的相同。初始化之后，在程序中可以看到中断和电机被启用，还有中断次数计数器 **int_cntr** 被设定了一个初始值。电机被设为以中等速度向前运行。

在中断服务程序的开始处，计数器 **int_cntr** 首先减 1。如果减 1 后的值为非零，退出中断服务程序。但是如果值为 0，那么表示时间过去了一秒钟，中断服务程序读取 Timer 0 和 Timer 1 的值并保存它们。频率测量就是以这种方式来完成的。在这个简单的演示程序中，我们没有对这个测量值做进一步的处理。但是在本章的后续部分我们将使用这个测量值来实现 AGV 的速度控制。

253

例程 9-5 在 AGV 中应用频率测量

```
;************************************************************************
;dbt_speed_meas
;Derbot wheel speed is measured by frequency counting. This is a demo
;program. The frequency measurement concept can be embedded into larger
;programs, eg for speed control.
;TJW 7.7.05                                              Tested 7.7.05
;************************************************************************
...
        #include p16f873A.inc
...
(opening program sections omitted)
...
;Initialise
;set up SFRs in Bank 1
main bcf status,rp1
...
        bsf     status,rp0      ;select memory bank 1
        movlw   B'11101000'     ;set up Timer 0: external input,
        movwf   option_reg      ;low to high transition, no prescale

        movlw D'249'            ;set PWM prd
        movwf   pr2
        bsf     pie1,tmr2ie     ;enable Timer 2 interrupt
;set up SFRs in Bank 0
        bcf     status,rp0      ;select bank 0
        movlw   B'00000011'     ;set up Timer 1: no prescale, oscillator
        movwf   t1con           ;disabled, external sync input
        movlw   B'01111100'     ;switch on Timer2, no prescale, /16 postscale
        movwf   t2con
        movlw   B'00001100'     ;select PWM
        movwf   ccp1con
        movwf   ccp2con
...
(further initialisation and startup omitted)
...
;clear timers
        clrf    tmr0
        clrf    tmr1l
        clrf    tmr1h
;enable interrupts
        movlw   D'250' ;load interrupt counter
        movwf   int_cntr
        bcf     pir1,tmr2if     ;clear pending interrupt
        bsf     intcon,peie     ;enable peripheral interrupt
        bsf     intcon,gie      ;enable global interrupt
;
;start running, then wait for Timer interrupt
        movlw   D'200' ;set PWM rate for reasonable forward speed
        movwf   ccpr1l
        movwf   ccpr2l
        bsf     portb,2         ;switch on optos
        bsf     porta,mot_en_left ;enable motors
        bsf     porta,mot_en_rt
```

```
wait goto wait
;
;*************************************************************************
;ISR is here. Interrupts occur every 4ms. Count 250 (i.e.1.0s), then
;measure pulse count on each wheel.
;*************************************************************************
Timer2_Int    bsf portc,5          ;diagnostic
       decfsz int_cntr
       goto   int_end
;here if making a measurement
       movf   tmr0,0 ;save counter values
       movwf  tmr0_temp
       movf   tmr1l,0
       movwf  tmr1_temp
       clrf   tmr0            ;clear counters
       clrf   tmr1l
       btfss  portc,0         ;but increment T1 if it is zero, as first
       incf   tmr1l                ;rising edge won't be seen
       movlw  D'250' ;reload interrupt counter
       movwf  int_cntr
int_end bcf pir1,tmr2if
       bcf    portc,5         ;diagnostic
       retfie
...
```

9.9 在 AGV 中应用速度控制

嵌入式系统是一个关于控制的系统，现在我们第一次应用一个闭合循环来进行速度控制。

例程 9-6 中的程序 **Dbt_speed_control** 使用刚讨论过的频率测量来实现 AGV 中电机的闭合循环的速度控制。它是基于一个简单的闭合循环控制系统[9.1]的工作原理：输出的变量(在例程中是指电机的速度，它是由轴角编码器的频率表示的)同要需要的值进行比较。这两者的差距被放大，然后使用放大后的值来驱动电机。对于每个电机来说，控制算法是相同的。算法在中断服务程序的注释行 **adjust left motor speed** 之后。

程序实际上是例程 9-5 的扩展，只是设定 **PR2** 寄存器(它设置了 PWM 的周期)为 255_D。这样做只是为了便于理解数据操作以及程序。但是它也将 Timer 2 的中断周期由 4.000ms 改变为 4.096ms，所以频率的测量就不再是精确的每半秒或者一秒一次。但是这个数据并不显示出来供人查看，因此并没有关系。

为了理解程序中的设置，我们首先必须要理解电机的速度与轴角编码器之间的关系。AGV 中使用的是一个每转产生 16 个脉冲的轴角编码器。当驱动电机的 L293 集成电路的电源为 9V 时，电机空转时变速箱的输出速度的测量值约为 154r. p. m。这个结果与表 A3-1 中的数据比较吻合，但是要注意由于驱动电路中的电压降，导致速度有一些下降。这个速度转化为一个最大的轴角编码器频率为 41Hz[(154×16)/60]。

为了对程序进行测试，电机的速度设置为最大速度的一半，这就意味着每 0.512s 轴角编码器将产生 10 个(近似整数)脉冲。这个值可以在程序中看到。

例程 9-6　AGV 中的速度控制程序

```
;******************************************************************
;dbt_speed_control
;Derbot wheel speed is measured by frequency counting. Derbot
;runs forward at fixed speed - demonstrates simple speed control.
;
;TJW 7.7.05                                          Tested, 24.7.05
;******************************************************************
...
       bsf      status,rp0                   ;select bank 0
...
       movlw    B'11101000' ;set up Timer 0: external input, increment
       movwf    option_reg   ; on low to high transition,no prescale
       movlw    0ff          ;set PWM prd to its maximum.
       movwf    pr2
       bsf      pie1,tmr2ie  ;enable Timer 2 interrupt
;set up SFRs in Bank 0
       bcf      status,rp0   ;select bank 0
       movlw    B'00000011'  ;set up Timer 1: no prescale, oscillator disabled,
       movwf    t1con        ;external input, sync with int clock
       movlw    B'01111100' ;switch on Timer2, no prescale, /16 postscale.
       movwf    t2con            ;Timer int prd is 4.096ms, freq is 244.14Hz
...
;******************************************************************
;ISR is here. Interrupts occur every 4.096ms. Count 125 (0.512s), then
;measure pulse count on each wheel, and calculate new PWM setting
;******************************************************************
Timer2_Int     bsf     portc,5      ;diagnostic
       decfsz int_cntr
       goto    int_end
;here if checking speed
       movf    tmr0,0 ;save counter values
       movwf   tmr0_temp
       movf    tmr1l,0
       movwf   tmr1_temp
       clrf    tmr0     ;clear counters
       clrf    tmr1l
       btfss   portc,0         ;but increment T1 if it is zero, as first
       incf    tmr1l              ;rising edge isn't seen
;Adjust left motor speed. Find "error" frequency by subtraction
       movf    tmr0_temp,0
       sublw   D'10'    ;this is demand speed
       movwf   tmr0_temp
       btfss   status,c ;check for polarity of result, skip if +ve
       goto    left_err_neg
       bcf     status,c ;here if result positive,
;"Amplify" result by shifting left.
;Check for over-range, bit 7 is initially 0 (from numbers used)
       btfss   tmr0_temp,6  ;check for possible over-range,
                           ;don't shift if bit 6 set
```

```
        rlf      tmr0_temp,f
        bcf      status,c
        btfss    tmr0_temp,6   ;shift again
        rlf      tmr0_temp,f
        bcf      status,c
        btfss    tmr0_temp,6   ;shift again
        rlf      tmr0_temp,f
        movf     tmr0_temp,w
        addwf    ccpr21,0      ;add in current PWM value, result to W
        movwf    tmr0_temp     ;test and correct for over-range
        btfsc    status,c
        goto     set_l_max     ;carry set, so set max op
        movf     tmr0_temp,0 ;output calculated result to PWM
        movwf    ccpr21
        goto     rt_adj        ;and proceed
set_l_max movlw       D'255' ;here if output value has saturated,
        movwf    ccpr21         ;hence output max value

        goto     rt_adj        ;and proceed
left_err_neg comf tmr0_temp,0 ;this will be holding a negative no,
        addlw 1                      ;therefore correct by taking 2's comp.
        movwf    tmr0_temp
        btfss    tmr0_temp,6   ;check for possible over-range,
        rlf      tmr0_temp,f    ;don't shift if bit 6 set
        bcf      status,c
        btfss    tmr0_temp,6   ;shift again
        rlf      tmr0_temp,f
        bcf      status,c
        btfss    tmr0_temp,6   ;shift again
        rlf      tmr0_temp,f
        movf     tmr0_temp,w
        subwf    ccpr21,1      ;and subtract from current PWM value.
                                  ;Over-range testing not included
;
;now adjust right motor speed
...
```

控制算法本身的流程如图 9-19 所示。可以依照这个流程来浏览程序代码。电机的速度在程序开始处设置为中间值(即零速度),后来由中断服务程序对它进行调整。程序只显示了左电机的控制代码,右电机的控制代码可以在本书附属资源的完整的程序代码中找到。

实际中程序运行很正常,这是由于控制电机的速度首先是从测量光敏传感器的输出波形频率中获得并通过调整后输出。当 AGV 在地面上行驶时,我们第一次看到它以相当完美的直线行驶,而不是当 2 个电机分别自由运转时,出现的那种逐渐弯曲的曲线。

需要注意的是,程序中从电机的速度设置到电机的速度测量和控制涉及的所有功能都是使用计数和定时技术来实现的。

257

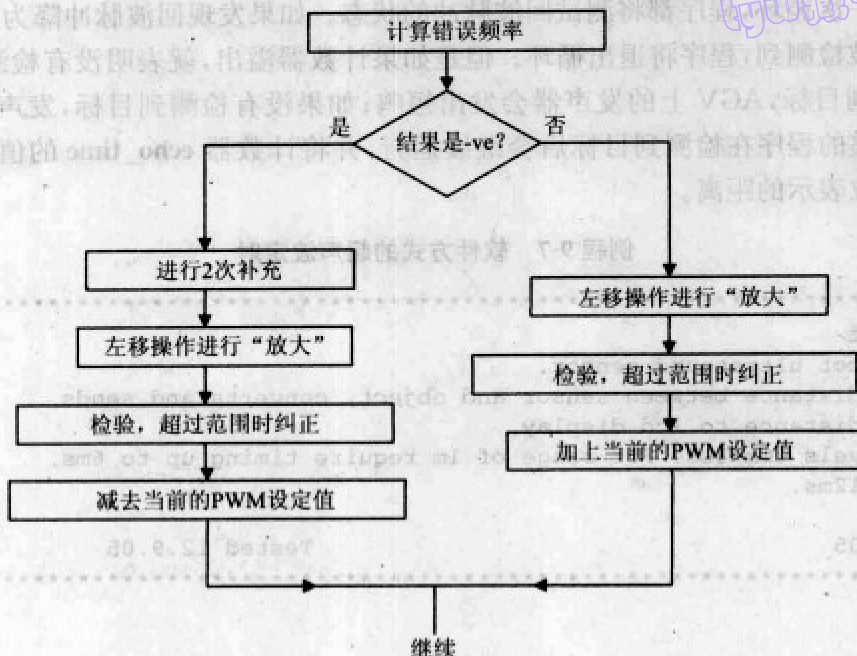


图 9-19　电机控制算法的流程图

9.10　当没有可用定时器时

尽管我们看到硬件定时器强大的功能，但是还是会出现没有可用定时器的情形。在微控制器中可能没有定时器或者它们可能全被占用。这种情况就会发生在 AGV 中，当所有的定时器都被占用，但是还要应用超声波传感器时，就需要另外一个定时器。这里采取的解决方案是使用第 5 章提到的软件定时。这种软件循环产生延时的方法有一个缺点：它完全占用 CPU 资源。但是在这里我们会发现它是一个有效的方法，由于超声波的测量并不是连续进行，因此测量时间占总时间的比例很小。

例程 9-7 显示了程序的部分代码。在 8.6.5 节中，它用于测量超声波传感器与其检测范围内的对象之间的距离。在第 11 章，这个距离将以厘米为单位显示在手动控制器的 LCD 显示板上，并且电路被修改成可进行超声波方式的距离测量。程序的第二部分没有列出来，这是因为其中的数据操作在第 11 章才会讲述。完整的程序代码在本书附属资源中可以看到。超声波传感器和诊断性 LED 共用引脚。因此，如果使用了传感器，就不能独立使用 LED，尽管它仍然会响应超声波的驱动脉冲而发光。

主程序应用了图 8-16 的时序图，它的结构是一个从标号 **us_loop** 开始的循环。超声波脉冲从端口 C 的位 5 发出。之后，程序延时一段时间来确保回波脉冲升起。然后，程序进入一段定时循环，每一次循环，计数器 **echo_time** 都会加 1。并且每一次循环的时间设计为精确的 30μs。声音的传播速度是 1mm/3μs，这就说明每一次循环声音将传播 10mm。由于声音必须传播到目标，然后才能返回，这就意味着测量的分辨率为

5mm。每一次循环，程序都将测试回波脉冲的状态。如果发现回波脉冲降为0，那么就说明目标被检测到，程序将退出循环。但是如果计数器溢出，就表明没有检测到目标。如果检测到目标，AGV上的发声器会发出短鸣；如果没有检测到目标，发声器会发出长鸣。完整的程序在检测到目标后会继续运行，并将计数器 **echo_time** 的值转换成以厘米为单位表示的距离。

例程 9-7 软件方式的超声波定时

```
;*************************************************************
;Dbt_US_tst
;Tests derbot ultrasound sensor.
;Measures distance between sensor and object, converts and sends
;measured distance to lcd display,
;Sound travels 1mm/3us. For range of 1m require timing up to 6ms,
;2m up to 12ms.
;
;TJW 10.9.05                                Tested 12.9.05
;*************************************************************
...
;
;Specify RAM
      cblock 20
...
echo_time     ;counter measuring time for echo to return
...
      endc
;
      org  00
      goto  begin
; (opening program sections omitted)
...
;**********************************
;The "main" program loop starts here
;**********************************
us_loop clrf  echo_time    ;clear counter used to measure time
      bsf   portc,5      ;output us pulse. 10us minimum required
      call  delay20u
      bcf   portc,5
      call  delay300u    ;pause for op to be set high; ie blank for 5cm
;this loop takes 30us per cycle, ie 10mm there and back, or 5mm one way;
;hence 8-bit range is 255x5mm = 1275mm. Max duration in loop is 30x255=7.6ms
echo  call  delay24u
      incf  echo_time,1
      btfsc status,z     ;test for overflow,which indicates no target found
      goto  no_tgt
      btfsc portc,6      ;test echo pulse, skip if cleared (ie echo recd.)
      goto  echo
;here if target detected
;     bsf   portb,1      ;indicate with short bleep
      call  delay200
      bcf   portb,1
      call  delay200
```

```
...
(data processing, and transfer to lcd inserted here)
...
        goto   us_loop
no_tgt  bsf    portb,1        ;indicate with long bleep
        call   delay500
        call   delay500
        bcf    portb,1
        call   delay200
        goto   us_loop
...
(most subroutines omitted)
...
;This subroutine introduces delay of 24us
delay24u movlw 6      ;6 cycles called, each taking 3us,
        ;plus call (2), & 2 opening insts (2) + 2 at end
        ;less one cycle lost when last goto is hopped
        movwf delcntr1
del21 decfsz delcntr1,1 ;3 inst cycles in this loop, ie 3us
        goto  del21
        nop
        return
```

建议你将这个程序暂时看作是一个理论讲解的例子，因为有一些必需的子例程在第 10 章和第 11 章才会介绍。

9.11 休眠模式

在第 6 章我们介绍了休眠模式，但是时间在那一章几乎是暂停的。16F87XA 的休眠模式的工作原理与 16F84A 的休眠模式一样，这在 6.6 节已经讲述过。16F84A 休眠模式的一个可能的限制是它唤醒的时机较少。缺少周期性定时唤醒就是一个很明显的例子。(尽管看门狗似乎可以提供这种功能，但是它的工作频率是不精确的，并且溢出周期的范围较窄。)

16F87XA 则提供了多种唤醒选择，包括许多来自外围设备的唤醒。图 7-10 的中断结构图表明所有的外围设备均可引发休眠后的唤醒。当然实际上有一些外围设备在休眠时会停止工作，由于它们依赖于时钟振荡器才能工作，而处于休眠模式时，时钟振荡器会停止振荡。一个有趣的事情是 Timer 1 可以使用自身的时钟振荡器来工作。当处于休眠模式时，Timer 1 将继续运行，这样就可以利用 Timer 1 的溢出中断来获得周期性唤醒。16F87XA 中可用的外围设备唤醒选择总结如下：

- ☐ Timer 1，当工作于异步模式时；
- ☐ CCP 设置为捕捉模式时产生的中断；
- ☐ 特殊事件触发，并且 Timer 1 在外部时钟下工作于异步模式；
- ☐ 同步串行端口；

- ❑ 可寻址的通用同步异步接收器发送器(Addressable Univeral Asynchronous Receiver Transmitter,USART)端口；
- ❑ 模数转换器，当工作于 RC 时钟振荡器时；
- ❑ EEPROM 的写完成；
- ❑ 比较器的输出改变；
- ❑ 对并行从动端口的读或写(16F874A 或者'877A)。

9.12 后面我们将学习什么

我们已经学习了一些非常有用的工具和技术，它们都是使用时基的应用开发。但是如同前面引入的大部分例子一样，这些技术都是以一种简单的形式呈现出来，似乎微控制器上仅仅就只处理这一件事。实际情况当然不是这样——AGV 上需要驱动电机，还要测量电机的速度，同时还要控制伺服等等。基于时间的任务会越来越多，它们之间可能会发生冲突。例如，我们已经看到一个定时器用于两个不同并且存在潜在冲突的任务。在嵌入式系统的应用中，调度这些不同时基的程序活动是很常见的。解决这个问题是一个挑战，需要一个系统化的解决方法。我们将在第 18 章中的实时操作系统(Real Time Operating System,RTOS)中看到这个系统化的方法。

9.13 AGV 硬件集成

为了运行本章使用的大部分的例程，你需要在 AGV 中增加反射性光学传感器，并且在车轮上增加轴角编码器。增加完这些器件后，AGV 电路将会进入如图 9-20 所示的阶段。为了运行例程 9-3，还需要增加一个伺服。这在图 9-20 中没有显示，但是在图 A3-1 中可以看到。

小结

- ❑ 定时是嵌入式系统设计的一个基本要素——它用于自身的定时和使其他嵌入的活动能够运作，比如串行通信和脉宽调制。
- ❑ 可用的定时器非常多，它们都有灵巧的附加功能，这使得它们可以用于捕捉、比较、产生周期性中断或者 PWM 脉冲流。
- ❑ 在复杂的应用中，微控制器中可能有多个定时器在同时工作，它们被用于一些不同并且可能存在冲突的任务。你可以对这个问题进行讨论：如何调度和协调许多不同时基的程序活动？

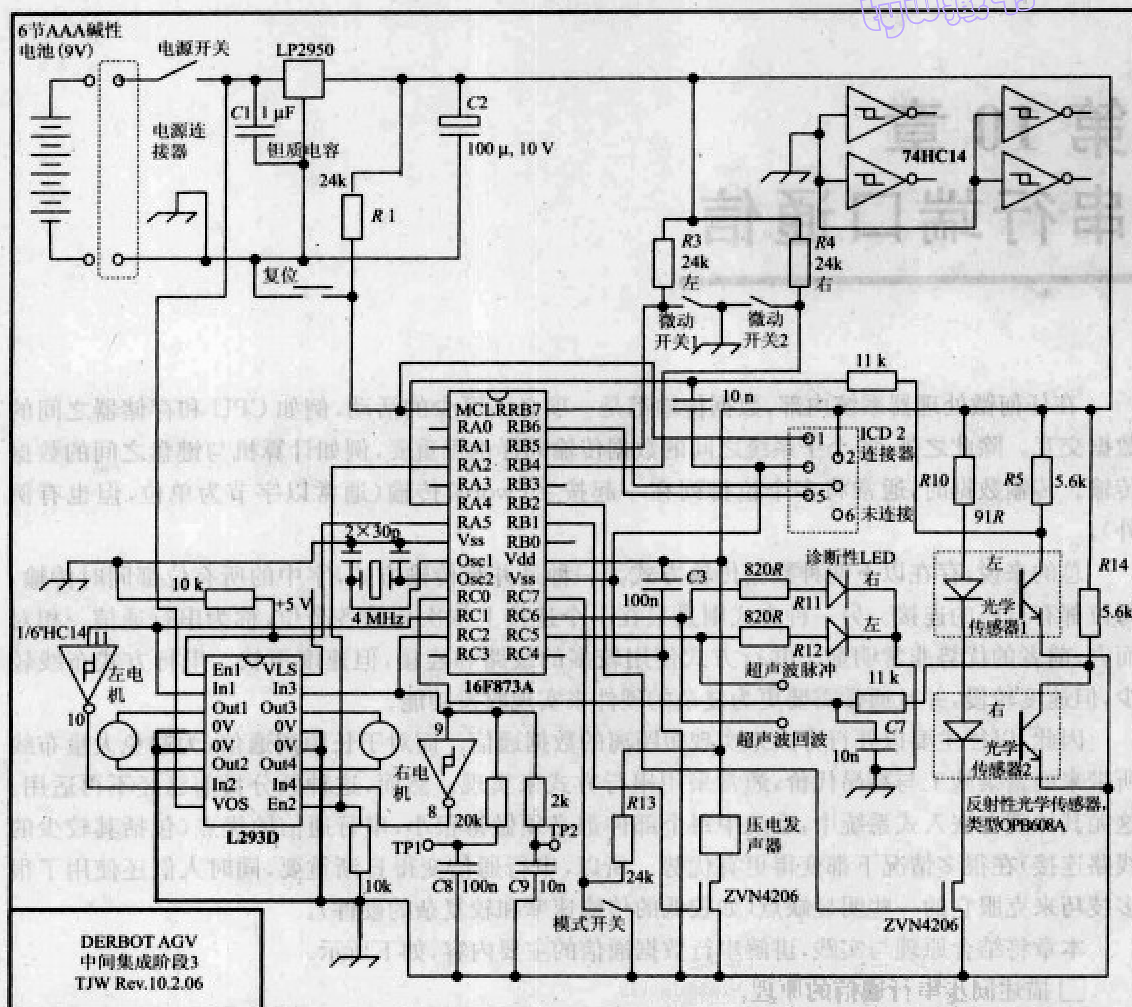


图 9-20 AGV 中间集成阶段 3

参考文献

9.1. Bolton, W. (1998). *Control Engineering Pocket Book*. Newnes, Oxford, UK. ISBN 0 75063928 8.

261
~
262

第 10 章 串行端口通信

在任何微处理器系统内部，数据传输都是一项必不可少的活动，例如 CPU 和存储器之间的数据交互。除此之外，各个子系统之间的数据传输同样也很重要，例如计算机与键盘之间的数据传输。传输数据时，通常将多个位排列在一起按字(word)传输(通常以字节为单位，但也有例外)。

总的来说，存在以下两种数据传输方式。一种是并行传输方式，字中的所有位都同时传输，每位都有自己的连接。另一种方式则是只在一个连接上依次传输各个位，称为串行通信。相对而言，前者的优势非常明显。并行方式使用较多的线路和连接，但速度更快。串行方式布线较少，但速度较慢，并且通常需要更为复杂的硬件来实现收发功能。

因此，以往主要以并行方式来实现短距离的数据通信。而对于长距离通信，为避免大量布线所带来的繁杂施工与高昂代价，通常采用串行方式来实现。然而，这种区分目前已经不再适用。这尤其体现在嵌入式系统中：系统中每个部件都必须做得很小，串行通信的优点(包括其较少的线路连接)在很多情况下都变得更有优势。所以，串行通信变得日渐重要，同时人们还使用了很多技巧来克服它的一些明显缺点(如较低的传输速率和较复杂的硬件)。

本章将结合原理与实践，讲解串行数据通信的主要内容，如下所示。

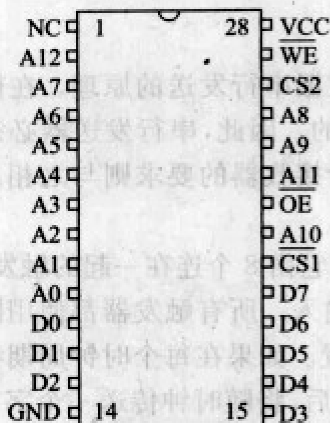
- 描述同步串行通信的原理。
- 以 PIC® 16F873A 为基础探索同步串行通信的实现方法，尤其是 SPI(Serial Peripheral Interface，串行外围设备接口)与 I^2C(Inter-Integrated Circuit，内置集成电路)总线协议。
- 描述异步串行通信的原理。
- 以 PIC 16F873A 为基础探索异步串行通信的实现方法。

与以往类似，本章中很多内容都通过示例代码的方式加以说明。这些代码是为 Derbot AGV 及其手动控制器面板所编写的，无论您是否拥有这个硬件来运行程序，都将发现很多乐趣。另外，本章还给出了一些串行端口数据的示波器跟踪波形。由于在编写本书时 MPLAB® 仿真器尚不支持串行端口，因此不能使用该仿真器来仿真本章中的程序。

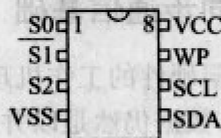
10.1 串行端口简介

上述引言刚刚强调了串行通信的一个主要优点，即它能够节省空间。图 10-1 就是一个例子。图中画出的 2 个存储器 IC 具有相同的存储容量。但是，具有并行端口连接的芯片需要 13 根地址线和 8 根数据线，这决定了芯片尺寸必然较大。使用串行端口连

接的芯片则只有 2 根用于传输数据和地址的连线(SCL 串行时钟、SDA 串行数据),以及 3 根地址选择线(S0,$\overline{S1}$,S2)。附带提一句,后者就是一个 I^2C 芯片,我们将很快在后面学到。它可以极大地节省 IC 级空间,减少 PCB 布线、带状电缆等。



(a) 并行地址与数据总线



(b) 串行地址与数据

图 10-1 2 种 8KB 存储器 DIP 封装的近似比例

虽然串行通信具有很多明显的优点,但必须认识到它所面临的挑战。由于仅使用一根线路传输数据,那么如何判断某位(bit)何时开始、何时结束,以及如何判断字的起止位置呢?对于这些问题,人们采用了很多有趣的解决方式。要识别各位可以采用以下 2 种方法。一种较为简单的方式是在发送数据的同时发送时钟信号,每个时钟周期标示一位数据,这种方式称为同步(synchronous)串行数据通信。另一种方式是不发送时钟信号,而在数据上加入某些时序的要求。这种情况下,无需时钟也可以实现通信,仍然可以有效地识别出数据位,这种方式称为异步(asynchronous)串行数据通信。本章将对这 2 种通信方式进行详细的讲解。

要识别整个字的起始和结束,通常需要将数据以某种格式进行打包。数据的同步与格式化意味着需要施加某些规则,用以确保连续一致的通信过程。这些规则被称为协议(protocol),是串行通信中的重要内容。某些协议相当简单,而另外一些则非常复杂。在本章中我们会学到部分协议。

在串行连接中,通常会用到*发送器*(transmitter)与*接收器*(receiver)的概念。前者是一种向串行连接输出数据的设备,后者是一种接收数据的设备。总之,有时将串行连接上的任何器件都称作*节点*(node)。

为了阐述串行通信的原理,我们将使用 16F873A 作为参考。第 7 章已经提到,它有 2 个串行端口,使用灵活,配置方式多样,因而也相当复杂。主同步串行端口(MSSP)适用于不同形式的串行通信,而可寻址的通用同步异步收发器(USART)则可运行于同步或异步模式下。

264

10.2 简单串行连接——同步数据通信

10.2.1 同步通信基础

认识底层硬件的工作机理有助于我们理解数据串行发送的原理。在微处理器或存储器内部，数据仍然是以并行格式放置并使用的。因此，串行发送器必须能够接受并行格式的数据，并以串行方式进行发送；而串行接收器的要求则与之相反。使用移位寄存器可以有效地实现这一功能。

图 10-2 显示了一个简单的 8 位移位寄存器。它由 8 个连在一起的触发器构成，每个触发器的 Q 输出端给出下一个触发器的数据输入。所有触发器都使用同一时钟驱动，数据在每个时钟周期向右移动一个触发器位置。如果在每个时钟周期都有一个新的数据位施加在 D_{IN} 输入端，那么 8 个时钟周期之后，将随时钟传送一个字节，此时，从输出端 Q_A 和 Q_H 就可以并行地读出一个字。这种简单的电路可以用作串行数据的接收器。

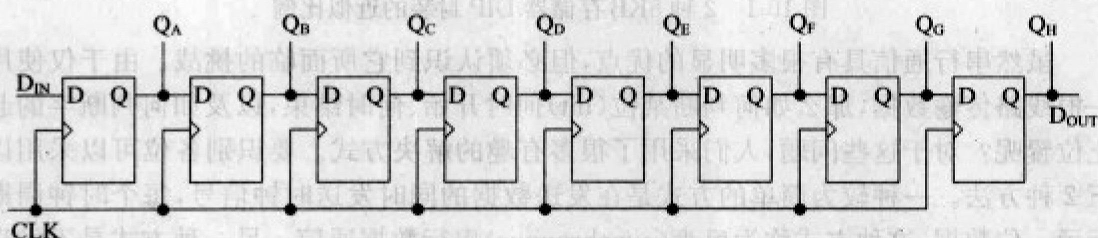


图 10-2 8 位移位寄存器——可用作串行数据接收器

要增强这个电路的功能并不困难：不仅可以从移位寄存器读出并行数据，还可以将并行字载入移位寄存器。可以使用模块框图来表示这种移位寄存器，如图 10-3 所示，而不必画出它的完整逻辑图。使用这种通用移位寄存器，可以随时钟将串行数据输入或输出，还可以将并行数据读出或载入。

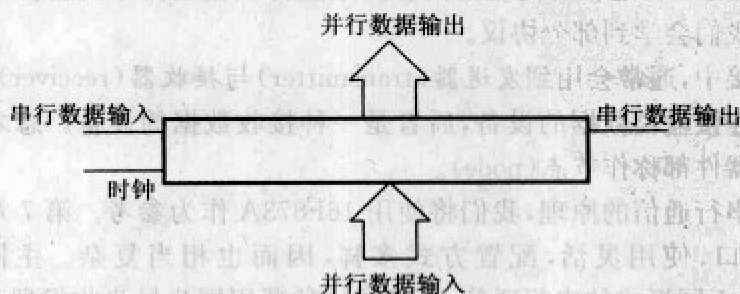


图 10-3 通用移位寄存器框图

现在我们已经了解了串行通信连接的基本原理。对于如图 10-3 所示的 2 个移位寄存器，如果利用图 10-4 所示的连接，并采用同一时钟源，那么一个移位寄存器的数据就可以串行地发送至另一个移位寄存器。通过将输出端返回连向输入端，2 个 8 位移位寄存器就构成了一个 16 位移位寄存器。8 个时钟周期之后，一个移位寄存器中的数据就移动到另一个移位寄存器中。因此，任何一个移位寄存器都可以看作发送器或接收器。数据传输动作实际上是由时钟控制的。由于数据传输与公共时钟信号同步，因此将这种连接称为同步连接。

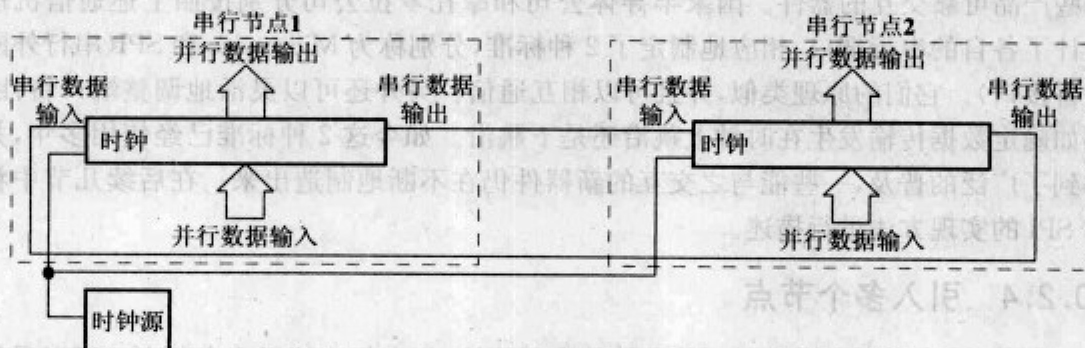
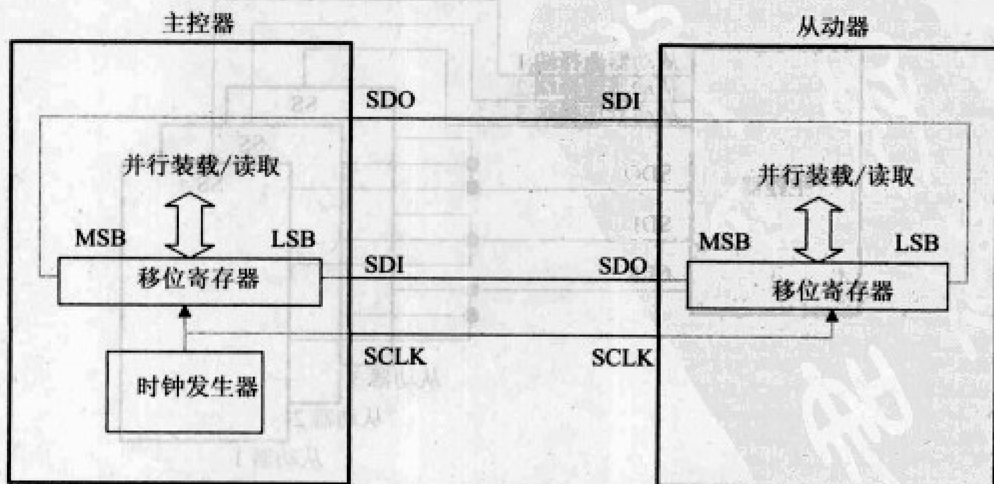


图 10-4 通用串行通信连接

10.2.2 在微控制器中实现同步串行 I/O

上述同步串行连接可以方便地在微控制器中实现，如图 10-5 所示。时钟源放置在微控制器内部，用于控制数据流，这种节点称为**主控器**(master)。另一个节点则称为**从动器**(slave)。从动器可以是微控制器、存储器或其他类型外围设备。



SDO(Serial Data Out, 串行数据输出) SDI(Serial Data In, 串行数据输入) SCLK(Serial CLocK, 串行时钟)

图 10-5 微控制器内部实现的同步串行连接

这类连接是很多简单嵌入式串行连接的一种。其中的移位寄存器有对应的存储器映射，可以对其进行读写操作。

10.2.3　Microwire 和 SPI

在 20 世纪 70 年代末和 20 世纪 80 年代初，美国国家半导体公司(NSC)、摩托罗拉公司以及其他一些公司致力于开发一种内建有同步串行通信能力的微处理器和微控制器。他们需要对器件运行特性加以定义，这样其他厂商就可以按照要求制造出能与这些产品可靠交互的器件。国家半导体公司和摩托罗拉公司分别按照上述通信机理设计了各自的串行端口，相应地制定了 2 种标准，分别称为 Microwire 和 SPI(串行外围设备接口)。它们的原理类似，并且可以相互通信。另外还可以灵活地调整端口特性，例如确定数据传输发生在时钟上跳沿还是下跳沿。如今这 2 种标准已经使用多年，并得到了广泛的普及，一些能与之交互的新器件仍在不断地制造出来。在后续几节中将对 SPI 的实现方法进行描述。

266

10.2.4　引入多个节点

图 10-5 的结构图描述了一种有效的串行连接，但它仅连接了 2 个节点。可以采用简单的方法加以扩展。如图 10-6 所示，通过引入一种方式使得主控器能够选择与之通信的从动器，可以将多个从动器连入串行数据线路。此时，需要利用使能端启用从动器的输入，称为*从动器选择端*(Slave Select, SS)或*片选端*(Chip Select)。在不同器件中，SS 线的确切功能会稍有不同，但或多或少地都具有将器件完全或部分地与串行连接断开的能力。此时，主控器必须为每个与之通信的从动器配备专用连接线，这可以用端口位输出来实现。

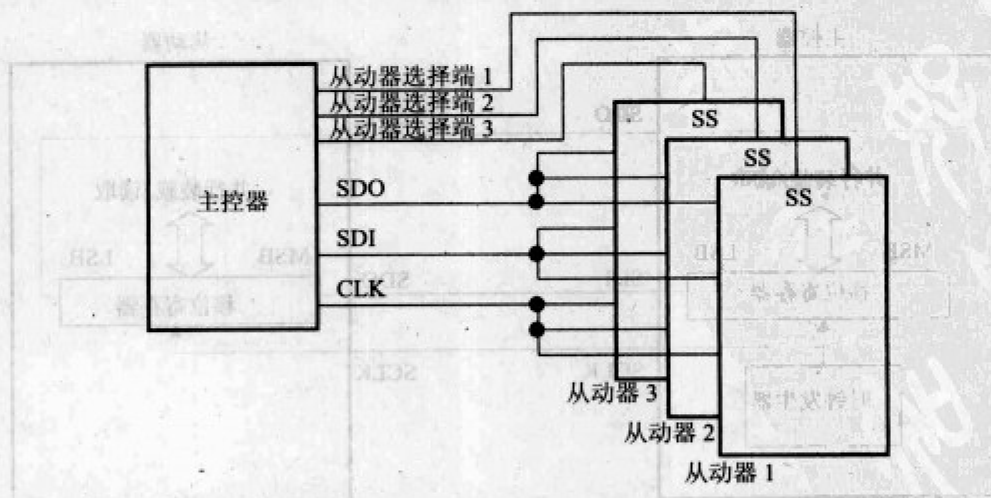


图 10-6　单个同步主控器与多个从动器之间的连接

10.3 16F87XA主同步串行端口(MSSP)模块的SPI模式

MSSP模块设计用于同步通信,并可配置为简单同步端口(称为SPI模式,但与SPI和Microwire都兼容)或I^2C端口。它具有3个专用SFR,包括SSPCON1、SSPCON2和SSPSTAT,它们可以在图7-6的寄存器文件映射图中找到。另外,它还有一个寄存器SSPBUF用于数据传输,并可作为中断源,如图7-10所示。本节将对该模块的SPI模式进行分析。

267

10.3.1 端口概述

图10-7描述了配置为SPI端口的MSSP。可以将其配置为主控器或从动器,作为主控器时可以选择不同的时钟速率。在下面的分析中将会看到它是如何基于上述简单串行概念而建立的。该串行端口的核心是移位寄存器SSPSR。在时钟驱动下,它将串行数据传送至SDO引脚(当输出缓冲门启用时),并从SDI引脚接收串行数据。如果

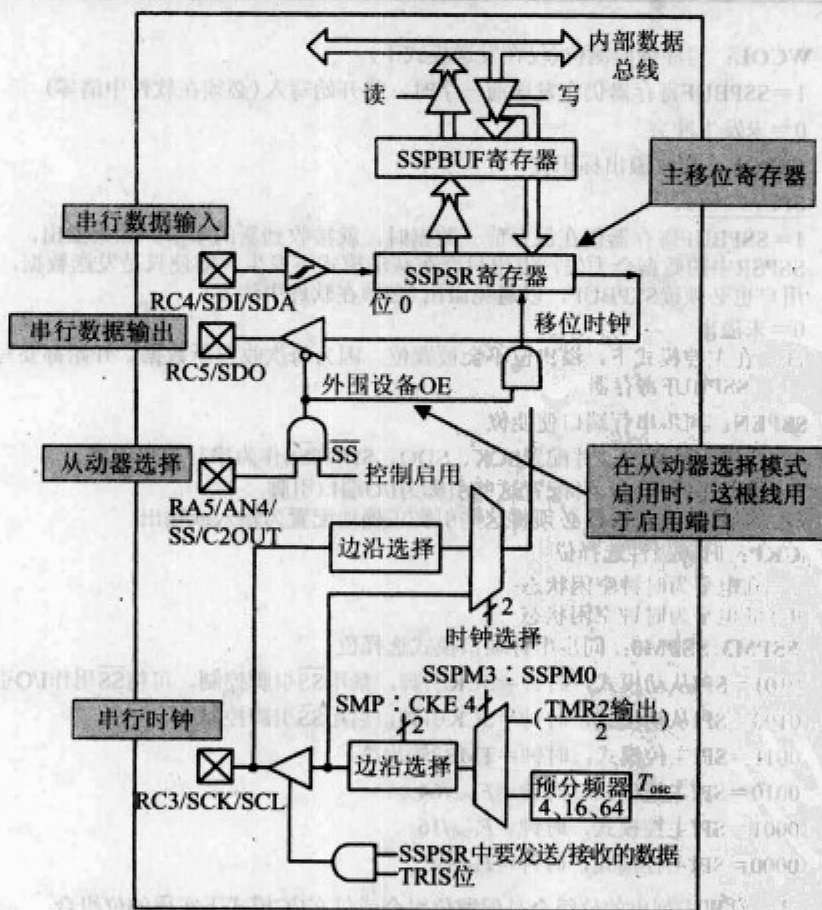


图10-7 SPI模式下的MSSP框图(阴影框中所附标签为作者所加)

端口被设置为从动器，它将通过 SCK 引脚接收来自系统主控器的时钟信号。如果端口被设置为主控器，它将产生时钟信号，并通过 SCK 引脚将其输出。这个时钟信号可以从内部时钟振荡器获得，也可以从 Timer 2 获得。

相对于前面提到的简单串行端口，这里有一个重要的增强特性，即添加了缓冲寄存器 SSPBUF。它在移位寄存器的进出通道上保存了 1 个数据字节，事实上是一个可由程序进行读写的可寻址寄存器。这些改进极大地提高了串行端口的灵活性。在移位寄存器运行过程中，可以向该缓冲器读写数据。例如，它可以临时保存 1 个接收字节，同时随时钟串行接收下一个字节。

10.3.2 端口配置

268

端口在 SPI 模式下的动作可以通过 SSPCON1 和 SSPSTAT 这 2 个寄存器来控制。它们的用法在 SPI 和 I²C 模式下有些不同。图 10-8 和图 10-9 分别列出了它们的配置

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
WCOL	SSPOV	SSPEN	CKP	SSPM3	SSPM2	SSPM1	SSPM0
位 7							位 0

位 7　**WCOL**：写冲突检测位(仅在发送模式下)

1=SSPBUF寄存器仍在发送前一字时，就开始写入(必须在软件中清零)

0=未发生冲突

位 6　**SSPOV**：接收溢出标识位

SPI从动模式

1=SSPBUF寄存器仍在保存前一数据时，就接收到新的字节。如果溢出，SSPSR中的数据会丢失。溢出只会在从动模式下发生。即使只是发送数据，用户也必须读SSPBUF，以避免溢出(必须在软件中清零)

0=未溢出

注：在主控模式下，溢出位不会被置位，因为每次收发新数据，开始都要写入SSPBUF寄存器

位 5　**SSPEN**：同步串行端口使能位

1=启用串行端口，并配置SCK、SDO、SDI和$\overline{SS}$作为串行端口引脚

0=禁止串行端口，并配置这些引脚为I/O端口引脚

注：当该位启用时，必须将这些引脚正确地配置为输入或输出

位 4　**CKP**：时钟极性选择位

1=高电平为时钟空闲状态

0=低电平为时钟空闲状态

位 3~0　**SSPM3**:**SSPM0**：同步串行端口模式选择位

0101=SPI从动模式，时钟=SCK引脚，禁用$\overline{SS}$引脚控制，可将$\overline{SS}$用作I/O引脚

0100=SPI从动模式，时钟=SCK引脚，启用$\overline{SS}$引脚控制

0011=SPI主控模式，时钟=TMR2输出/2

0010=SPI主控模式，时钟=$F_{OSC}/64$

0001=SPI主控模式，时钟=$F_{OSC}/16$

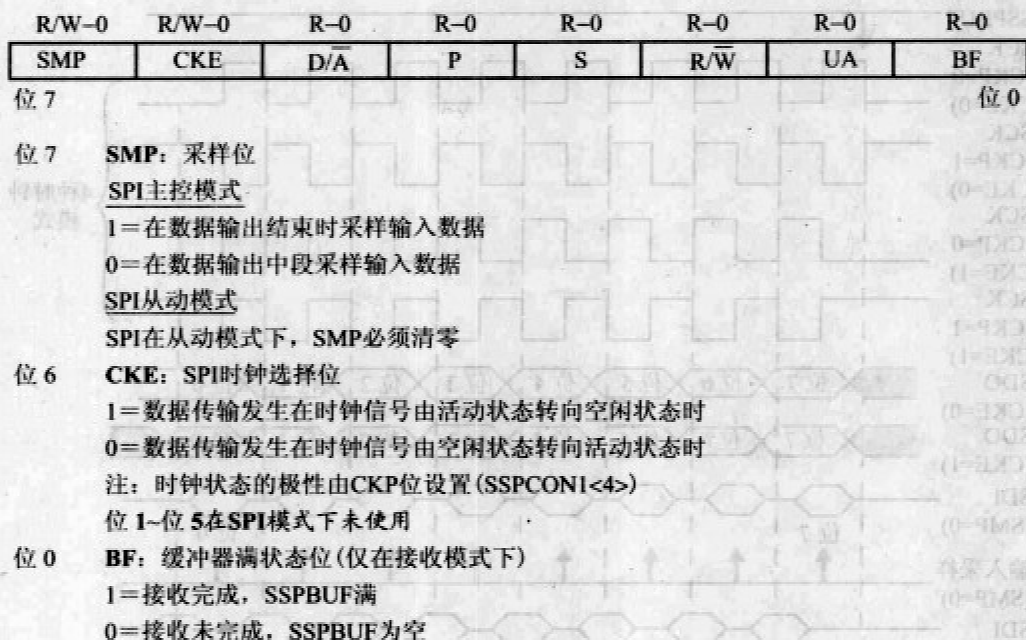
0000=SPI主控模式，时钟=$F_{OSC}/4$

注：这里未列出的位组合是保留位组合或仅在I²C模式下实现的位组合

图 10-8　SPI 模式下的 SSPCON1 寄存器(地址 14_H)

方法。其中的某些位可以实现以下功能：

- ☐启用并配置端口；
- ☐设置时钟频率与时钟特性；
- ☐管理数据传输与缓冲。



R/W-0	R/W-0	R-0	R-0	R-0	R-0	R-0	R-0
SMP	CKE	D/$\overline{A}$	P	S	R/$\overline{W}$	UA	BF

位 7 位 0

位 7 **SMP**：采样位

SPI主控模式

1＝在数据输出结束时采样输入数据

0＝在数据输出中段采样输入数据

SPI从动模式

SPI在从动模式下，SMP必须清零

位 6 **CKE**：SPI时钟选择位

1＝数据传输发生在时钟信号由活动状态转向空闲状态时

0＝数据传输发生在时钟信号由空闲状态转向活动状态时

注：时钟状态的极性由CKP位设置(SSPCON1<4>)

位 1~位 5在SPI模式下未使用

位 0 **BF**：缓冲器满状态位(仅在接收模式下)

1＝接收完成，SSPBUF满

0＝接收未完成，SSPBUF为空

图 10-9 SPI 模式下的 SSPSTAT 寄存器(地址 94_H)

SSPCON1 中的位 5(SSPEN)用于启用端口，低 4 位用于选择运行模式。这几位用于确定端口是作为主控器还是从动器。主控器模式下有 4 种时钟源可选。如第 9 章中图 9-4 所示，这几个时钟源来自于振荡器信号的 4 分频、16 分频或 64 分频，或者 Timer 2 的输出。

如果处于从动器模式，通过设置 SSPCON1 的低 4 位就可以启用从动器选择(Slave Select)输入引脚$\overline{SS}$。此时，外部$\overline{SS}$信号可以控制驱动 SDO 引脚的三态缓冲器以及送入移位寄存器的时钟信号。这样，$\overline{SS}$输入就可以有效地启用串行端口动作，并将端口用于多节点系统结构中，如图 10-6 所示。 269

对于发生数据或时钟传输的所有引脚，必须根据需要设置数据方向位。因此，对于与端口 C 的位 5 复用的 SDO 引脚，必须将 TRISC 的位 5 清零从而将该引脚设置为输出。同样地，对于 SCK 引脚，在主控模式下必须将 TRISC 的位 3 清零(设为输出)，而在从动模式下将其置位(设为输入)。而 SDI 引脚则完全处于 MSSP 模块的直接控制之下。

10.3.3 时钟设置

图 10-10 显示了模块处于主控模式下的时钟波形与数据波形的关系。当 CKP 位被 270

置1时,时钟空闲状态为逻辑1。发生数据传输的时钟边沿类型由CKE位决定。对于到来数据,数据采样时刻由SMP位决定。这些位的具体设置方式取决于所用从动器件的特定需要。当然,在单个互连系统内部始终保持这些设置的一致性是非常重要的。

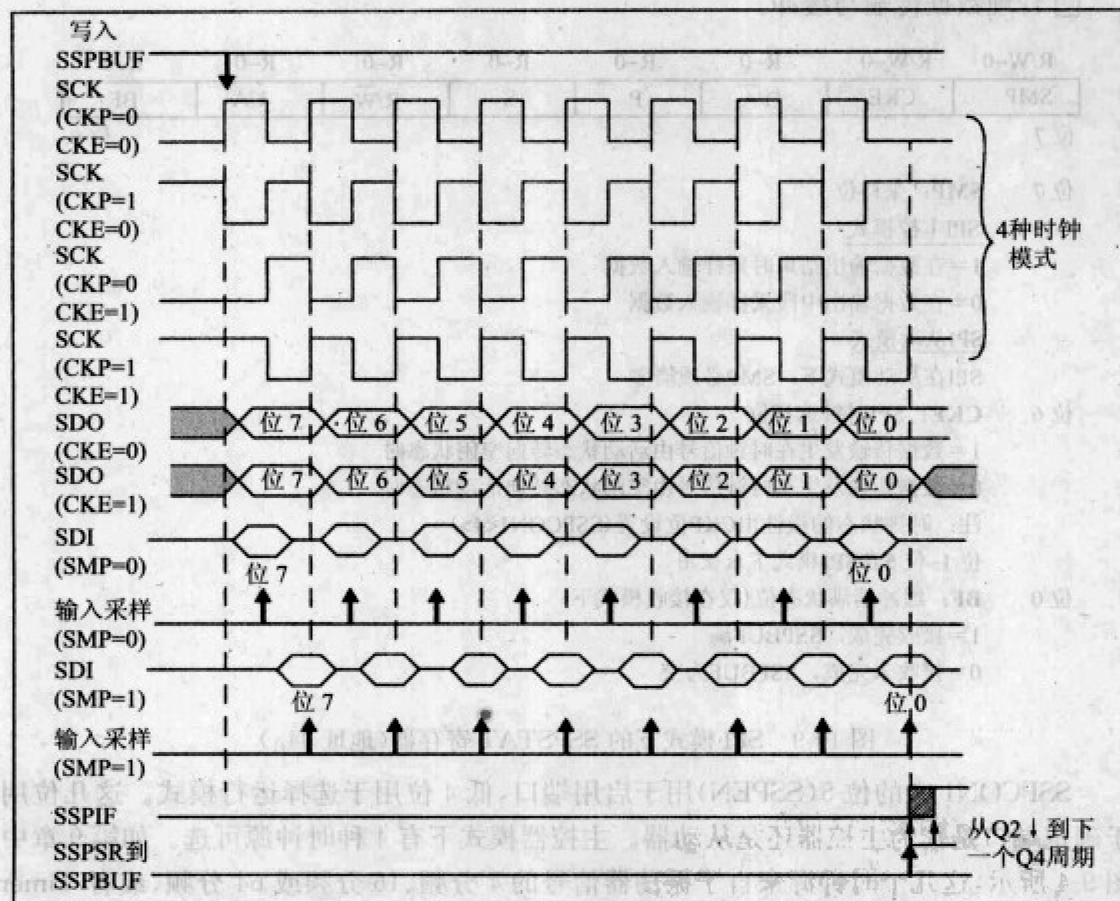


图 10-10 主控模式下的SPI时序图

10.3.4 管理数据传输

同步串行端口(例如现在正分析的这种端口)可以设置为主控器或从动器。无论哪种模式,都可以通过应用软件将其用作接收器或发送器。无论哪种应用,数据总是从移位寄存器的一端随时钟输出并进入另一个移位寄存器。具体使用哪种数据由用户决定。

因此,在串行端口使用过程中,需要考虑时序和控制方面的问题。如果端口被设置为从动器,那么数据传输将由外部设备发起,从动器的端口必须对此做出灵敏的反应。它必须根据数据传输方向将数据移入或移出端口缓冲器。如果被设为主控器,数据传输由端口发起,它必须也能够向缓冲器移入或移出数据。在任何一种情况下,串行端口的硬件都必须能够在程序执行其他无关任务的同时执行数据传输。为此,端口

提供了 SFR 中的一些状态位以及中断来辅助管理这一过程。 271

如果将端口设置为主控器，一旦向缓冲寄存器 SSPBUF 写入数据，将自动启动数据传输，随时钟将 SSPBUF 中装载的数据逐位输出，并将 SDI 引脚上的数据逐位输入。8 个时钟周期之后，中断标志 SSPIF 将被置位，移位寄存器 SSPSR 中的数据将自动传送至缓冲器 SSPBUF。SSPIF 标志可用作中断，提醒 CPU 传输已结束。如果在前一个字发送完毕之前，就向 SSPBUF 重新写入，那么将置位写冲突位 WCOL。

如果将端口设置为从动器，当 SCK 输入端开始状态切换时，端口将通过 SDI 引脚将数据随时钟输入 SSPSR 移位寄存器。同时，端口通过 SDO 引脚将寄存器另一端的数据随时钟输出。当然，系统设计员可以根据具体需要，在 SSPSR 寄存器和 SDI 输入脚上放置有效数据。当 8 个时钟周期完成之后，中断位 SSPIF 将被置位，并且 SSPSR 中的数据将自动传送至缓冲器 SSPBUF。如果上一字节尚未从 SSPBUF 中读取，那么 SSPOV 位将被置位，表明接收溢出。 272

10.4 SPI 的简单例子

本书中所描述的 Derbot AGV 版本并没有使用 SPI 数据通信。但是，例程 10-1 所给出的简单例程可以在 Derbot 硬件上运行。与前面一样，程序的完整列表显示在本书附属资源中，下述例程只给出了一些与此直接相关的内容。程序非常简短，例程几乎给出了所有代码。参照图 10-8 和图 10-9 中的控制寄存器，可以对程序中的初始化设置进行对照检查。

程序通过写 SSPCON1（为实现向上兼容，汇编包含文件将其称作 SSPCON）启用 Derbot 微控制器的 SPI 模式，并将其设为主控器，时钟频率设为 $F_{OSC}/16$。这里将时钟控制位 SMP 和 CKE 都设为 0。通过 TRISC 将时钟输出引脚和数据输出引脚都设为输出模式。这样，程序将在串行连接上依次重复传输 2 个字节，字节间隔 40μs。

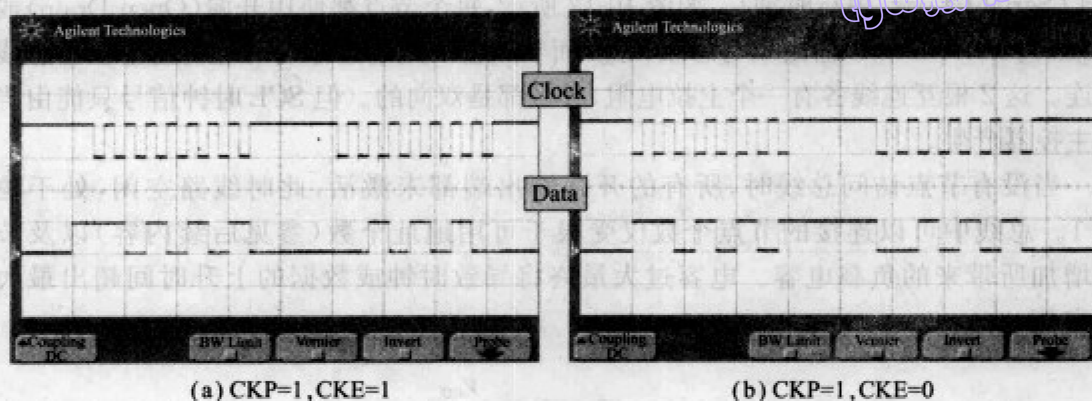
例程 10-1　SPI 示范程序

```
;*********************************************************************
;sync_ser_demo
;Program to demonstrate MSSP serial output.
;Program sends same two digits repeatedly from serial port, with delay.
;serial data appears on Port C bit 5, serial clock on Port C bit 3.
;2.7.05  TJW                                      Tested 2.7.05
;*********************************************************************
...
(early comments and initialisation omitted)
...
;
        bsf     status,rp0      ;select memory bank 1
...
        movlw   B'10000000' ;Set port C bits, SDO and SCK set as op.(SDI line,
        movwf   trisc           ;bit 4, is controlled by SPI module, so leave)
```

```
...
        bcf      status,rp0     ;select memory bank 1
        movlw    B'00000000'
        movwf    sspstat        ;SMP=0, CKE=0, other bits don't apply
        movlw    B'00110001' ;enable serial port, master mode, clock is fosc/16
        movwf    sspcon         ;& idles high.
;Switch all  outputs off
        clrf     porta
        clrf     portb
        clrf     portc
loop    movlw    B'11010101'
        movwf    sspbuf
        call     delay40u
        movlw    B'00101010'
        movwf    sspbuf
        call     delay40u
        goto     loop
;
;Subroutine: introduces delay of 40us approx
delay40u   movlw      D'10'     ;10 cycles called, each taking 4us
        movwf   delcntr1
del1    nop                     ;4 inst cycles in this loop, ie 4us
        decfsz delcntr1,1
        goto    del1
        return
        end
```

图 10-11 显示了例程 10-1 中数据线与时钟线的示波器跟踪波形，分别与 CKE=1 和 CKE=0 这两种情况对应。这些波形可看作图 10-10 中某些波形的实例验证。水平分辨率设置为每格 10μs，这样可以精确地看到时钟周期为 4μs，也即频率为 250kHz。这对应于 SSPCON1 中的时钟设置 $F_{OSC}/16$（其中 F_{OSC} 为 4MHz）。从中还可以清楚地看到 2 个数据字节 11010101 和 00101010，数据传送时首先传送数据的 MSB。两种情况下的 CKP 位（在 SSPCON1 中）都设为高电平，因此图中时钟空闲状态为逻辑 1。当 CKE 位（在 SSPSTAT 中）被设为高电平时，可以看到输出数据在时钟的正跳变信号沿上发生改变，也即时钟信号从“活动”状态（此例中为逻辑 0）转向空闲状态时传输数据。当 CKE 为 0 时，情况则正好相反。注意，数据线的空闲状态是不确定的。

字节之间的时序也很有趣。注意程序在发送字节之间调用了 40μs 的延时，但图中的时间间隔却小于 20μs。这是因为，数据传输过程是由程序向 SSPBUF 的写入动作所发起的。然后程序将调用延时子例程，该子例程的执行过程与数据传输过程同时进行。



(a) CKP=1,CKE=1　　(b) CKP=1,CKE=0

图 10-11　同步串行输出

10.5　Microwire 和 SPI 以及简单同步串行传输的局限性

从上述描述可以看出，Microwire 和 SPI 等同步连接提供了一种简单可靠的数据连接，但是它们依然存在某些局限性，包括：

- ❑ 它们不能很好地适应需要多主控器的情况；
- ❑ 它们不能寻址；
- ❑ 它们没有应答机制——发送器不能确定信息是否已被接收；
- ❑ 它们依然不够灵活——额外添加一个节点并不容易，即便只是添加一个从动器也很不方便，因为每添加一个新的从动器，都需要另加一根从动器选择线（至少对于图 10-6 中的结构是这样的）。

10.6　增强的同步串行通信及芯片间总线

I^2C 总线协议是一种由非利普公司开发的串行通信标准，可以克服 Microwire 或 SPI 的某些缺陷。顾名思义，它用于提供单个系统内各个 IC 之间的通信。它最初的设计目标在于灵活的通信方式，以及兼容不同技术和传输速度的能力。与所有优秀标准一样，该标准已得到了非常广泛的应用，并超出了最初设定的应用范围。参考文献 10.1 对 I^2C 规范给出了完整的描述，参考文献 10.2 则对其进行了有效的注解。

10.6.1　I^2C 的主要特性与物理连接

与 SPI 和 Microwire 类似，I^2C 基于节点间的主—从关系。主控器控制所有的总线使用。I^2C 总线有 3 种不同的模式：标准模式（时钟频率达 100kHz）、快速模式（时钟频率达 400 kHz）和高速模式（最大时钟速率为 3.4Mb/s）。

I^2C 总线仅使用 2 根线路实现所有连接，分别称为 SDA（Serial DAta，串行数据）和

SCL(Serial CLock,串行时钟)。如图10-12所示,每个节点都使用开漏(Open Drain)或开集(Open Collector)输出端与总线相连,而节点输入则通过标准的逻辑缓冲器与总线相连。这2根互连线各有一个上拉电阻,并且都是双向的。但SCL时钟信号只能由当前主控器产生。

当没有节点访问总线时,所有的开漏输出端都未激活,此时线路空闲,处于逻辑1。总线中可以连接的节点个数仅受限于可用地址个数(参见后续内容)以及节点增加所带来的负载电容。电容过大最终将导致时钟或数据的上升时间超出最大额定值。

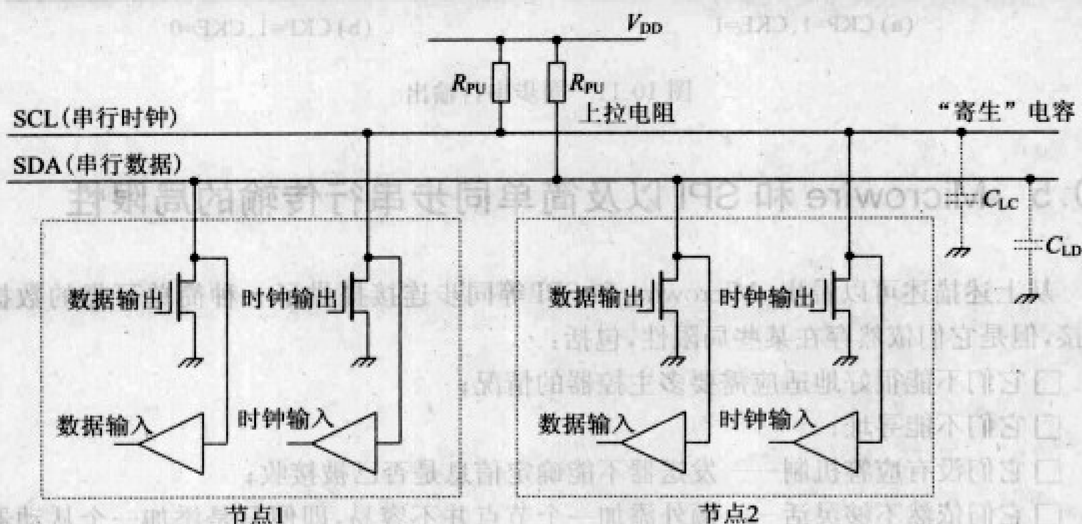


图 10-12 I²C 互连基础

10.6.2 上拉电阻

由于2根线都没有有源上拉,因此上拉电阻和线路电容一起决定了上升时间。规范要求最大线路电容不超过400 pF,标准模式下的最大上升时间(从逻辑0到逻辑1)为1000 ns。可以根据线路电容值选择合适的电阻值以达到上述时间要求。电阻值越低,上升时间越小,但电流消耗量会更大。通常使用4.7 kΩ的电阻值。如果线路电容较大,需要适当调低电阻值,反之当线路电容较小时则可以增加电阻值。参考文献1.1给出了相关的计算示例。

10.6.3 I²C 信号特性

I²C协议在数据传输过程中遵循非常清晰的格式,如图10-13所示。数据传输由启动(Start)条件所发起,即SCL线维持高电平时,SDA线跳变到低电平。数据传输最终由停止(Stop)条件所终止,即时钟维持高电平时SDA线跳变为高电平。与时钟信号类似,启动和停止条件都由当前主控器发起。

在启动与停止之间,数据按字节传输。在传输过程中,只有当SCL为低电平时

SDA值才能改变；当SCL为高电平时，数据应维持不变。这样，数据传输就可以正常进行，并能够识别出启动与停止信号。任何传输的第1个字节都包含地址信息。标准中允许使用7位地址(占用1个字节)或10位地址(占用2个字节)。图中显示的是7位地址。无论是哪种模式，第1个字节的第8位都是读/写位，该位用于确定后续信息的数据流向。在每个字节的最后，发送器都将释放SDA线，接收器都必须发送一个应答位将SDA线拉低。1条信息中(即启动位与停止位之间)可以包含任意数量的字节。

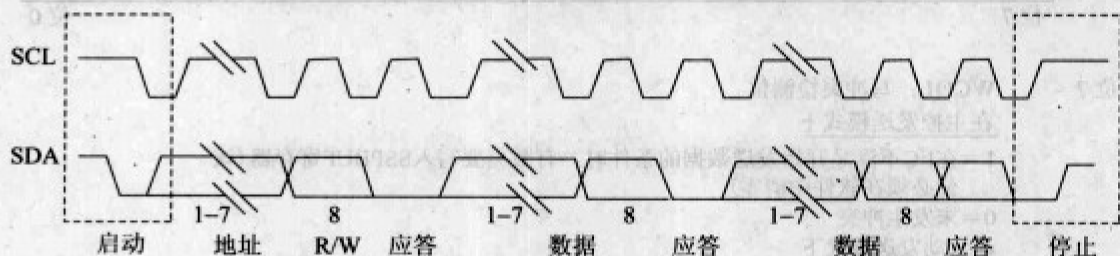


图 10-13　I²C信号特性

在上述方式之外有2个例外。通信中可以使用"全局呼叫(general call)"地址，所有地址位都为0。该地址用于同时寻址总线上的所有节点。如果在一次通信即将完成时，主控器又要启动新的通信，那么可以使用"重启动(Repeated Start)"信号。

I²C的一个重要特性就是允许存在多个主控器，并且节点可以在从动模式和主控模式之间进行切换。当总线空闲时，任何潜在的主控器都可以接管控制总线。如果2个节点同时试图控制总线，此时将使用仲裁方法，详见参考文献10.1。

10.7　配置为I²C的MSSP

16F873A中的MSSP可以配置为I²C模式。此时，SCL与端口C的第3位复用，SDA与端口C的第4位复用。在I²C模式下，端口相对于SPI模式更为复杂，需要认真分析加以理解。端口复杂性首先体现在新增的控制寄存器SSPCON2上，在I²C模式下需要使用它。

下面将循序渐进地讲解这种复杂却有趣的串行通信的应用，并使用AGV程序加以说明。

10.7.1　MSSP中的I²C寄存器及其基本应用

在SPI模式下的MSSP中，模块硬件的核心寄存器是移位寄存器SSPSR和缓冲寄存器SSPBUF。I²C模式则添加了地址寄存器SSPADD。它用于在从动模式下保存从动器地址，而在主控模式下则构成波特率发生器的一部分。后面陆续给出了从动模式和主控模式下的模块硬件结构图。

MSSP在I²C模式下使用的2个控制寄存器SSPCON1和SSPSTAT已经在前面

介绍过了。但是，这里的多数控制位的功能已经发生了变化，因此从学习角度来讲，完全可以将它们看作不同的SFR。图10-14和图10-15给出了相关描述。为满足I²C更复杂的控制要求，新加了控制寄存器SSPCON2，如图10-16所示。这样，在I²C操作中，程序员总共需要直接使用6个寄存器，包括与端口C和中断有关的寄存器。

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
WCOL	SSPOV	SSPEN	CKP	SSPM3	SSPM2	SSPM1	SSPM0
位7							位0

位7　**WCOL**：写冲突检测位

在主控发送模式下

1＝在I²C不满足开始发送数据的条件时，有数据要写入SSPBUF寄存器（该位必须在软件中清零）

0＝未发生冲突

在从动发送模式下

1＝仍在发送前一字时又有数据写入SSPBUF寄存器（该位必须在软件中清零）

0＝未发生冲突

在接收模式下（主控或从动模式）

该位被忽略

位6　**SSPOV**：接收溢出标识位

在接收模式下

1＝当SSPBUF寄存器中仍保存前一字节时接收到下一字节（必须在软件中清零）

0＝未溢出

在发送模式下

发送模式下该位被忽略

位5　**SSPEN**：同步串行端口启用位

1＝启用串行端口，并配置SDA和SCL作为串行端口引脚

0＝禁止串行端口，并配置这些引脚为I/O口引脚

注：当该位启用时，必须将SDA和SCL引脚正确地配置为输入或输出

位4　**CKP**：SCK释放控制位

在从动模式下

1＝释放时钟

0＝保持时钟信号为低电平（时钟延长）。（用于保证数据建立时间）

在主控模式下

在此模式下未使用

位3~0　**SSPM3**:**SSPM0**：同步串行端口模式选择位

1111＝I²C从动模式，10位地址，允许启动位中断和停止位中断

1110＝I²C从动模式，7位地址，允许启动位中断和停止位中断

1011＝I²C固件控制的主控模式（从动空闲）

1000＝I²C主控模式，时钟＝$F_{OSC}/(4*(SSPADD+1))$

0111＝I²C从动模式，10位地址

0110＝I²C从动模式，7位地址

注：这里未列出的位组合是保留位组合或仅在SPI模式下实现的位组合

图10-14　I²C模式下的**SSPCON1**寄存器（地址14_H）

R/W-0	R/W-0	R-0	R-0	R-0	R-0	R-0	R-0
SMP	CKE	$D/\overline{A}$	P	S	$R/\overline{W}$	UA	BF
位 7							位 0

位 7　SMP：转换率控制位
在主控或从动模式下
1＝在标准速率模式下(100 kHz和1MHz)禁用转换率控制
0＝在高速模式下(400 kHz)，启用转换率控制

位 6　CKE：SMBus选择位
在主控或从动模式下
1＝启用SMBus特定输入
0＝禁止SMBus特定输入

位 5　$D/\overline{A}$：数据/地址位
在主控模式下
保留
在从动模式下
1＝表明接收或发送的最后字节是数据
0＝表明接收或发送的最后字节是地址

位 4　P：停止位
1＝表明最近检测到停止位
0＝表明未检测到停止位
注：复位时和SSPEN清零时，该位清零

位 3　S：启动位
1＝表明最近检测到启动位
0＝表明未检测到启动位
注：复位时和SSPEN清零时，该位清零

位 2　$R/\overline{W}$：读/写位信息(仅在I²C模式下)
在从动模式下
1＝读
0＝写
注：该位用于保存最后一次地址匹配后的$R/\overline{W}$位信息。仅从本机地址与接收地址匹配开始，到下一个启动位、停止位或无$\overline{ACK}$位时，该位有效
在主控模式下
1＝正在进行发送
0＝未进行发送
注：该位与SEN、RSEN、PEN、RCEN或ACKEN进行"或"运算的结果表示MSSP是否处于空闲模式

位 1　UA：更新地址(仅在10位从动模式下)
1＝表明用户需要更新SSPADD寄存器中的地址
0＝不需要更新地址

位 0　BF：缓冲器满状态位
在发送模式下
1＝接收完成，SSPBUF满
0＝接收未完成，SSPBUF为空
在接收模式下
1＝数据正在发送(不包括$\overline{ACK}$位和停止位)，SSPBUF满
0＝数据发送完毕(不包括$\overline{ACK}$位和停止位)，SSPBUF空

图 10-15　I²C模式下的SSPSTAT寄存器(地址 94_H)

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
GCEN	ACKSTAT	ACKDT	ACKEN	RCEN	PEN	RSEN	SEN
位 7							位 0

位 7　**GCEN**：全局呼叫使能位（仅在从动模式下）

1＝SSPSR接收到全局呼叫地址(0000h)时启用中断

0＝禁止全局呼叫地址

位 6　**ACKSTAT**：应答状态位(仅在主控发送模式下)

1＝没有收到来自从动器件的应答

0＝收到来自从动器件的应答

位 5　**ACKDT**：应答数据位(仅在主控接收模式下)

1＝不应答

0＝应答

注：用户在接收结束时启动一个应答序列，同时发送该值

位 4　**ACKEN**：应答序列使能位(仅在主控接收模式下)

1＝在SDA和SCL引脚启动应答序列，发送ACKDT数据位，由硬件自动清零

0＝应答序列空闲

位 3　**RCEN**：接收使能位(仅在主控模式下)

1＝启用I²C接收模式

0＝接收空闲

位 2　**PEN**：停止条件使能位(仅在主控模式下)

1＝在SDA和SCL引脚发起停止条件，由硬件自动清零

0＝停止条件空闲

位 1　**RSEN**：重新启动条件使能位(仅在主控模式下)

1＝在SDA和SCL引脚发起重新启动条件，由硬件自动清零

0＝重新启动条件空闲

位 0　**SEN**：启动条件使能/延长使能位

在主控模式下

1＝在SDA和SCL引脚发起启动条件，由硬件自动清零

0＝启动条件空闲

在从动模式下

1＝为从动发送和从动接收启用时钟延长(已启用延长)

0＝仅为从动发送启用时钟延长(兼容PIC 16F87X)

280

图 10-16　I²C 模式下的 SSPCON2 寄存器(地址 91_H)

与 SPI 模式类似，通过设置 SSPCON1 寄存器中的 SSPEN 位可以启用 MSSP 的 I²C模式。运行模式(无论主控模式还是从动模式)以及地址长度都由 SSPCON1 中的低 4 位设置所决定。由图 10-14 可以看出存在 6 种可能的 I²C 运行模式。

SSPSTAT 寄存器位给出了端口当前状态的大部分信息，新的 SSPCON2 寄存器(图 10-16)则用于发起各种 I²C 动作。例如，置位 SEN 将发起启动信号，置位 PEN 发起停止信号，置位 RSEN 则发起重新启动信号。下面很快会看到相关示例。

要深入了解这些位的使用方法及其时序，或多或少都需要研究器件数据表中给出的时序图。时序图有很多，每一个都对应一种运行模式。本章稍后将对其中的 2 种运

行模式下的时序进行讲解。在很大程度上，针对 MSSP 的 I²C 模式的软件开发技术就是如何使这些时序图得到很好满足的过程。这并不意味着图中列出的每一位都必须用到，有时只需使用其中的一部分。但是，必须遵守图中描绘的事件流程。这些图看起来并不简单，因此在很多情况下只需使用已有的软件或对其稍做修改，而不是从头进行编写。

10.7.2 I²C 从动模式下的 MSSP

当配置为 I²C 从动模式时，MSSP 按照图 10-17 所示的方式运行。该图并不复杂，并且与图 10-7 中的 SPI 模式图有很多相似的地方。其核心部件依然是移位寄存器 SSPSR 和缓冲器 SSPBUF。移位寄存器的时钟由外部引脚 SCL 提供，数据的移入或移出通过引脚 SDA 完成。

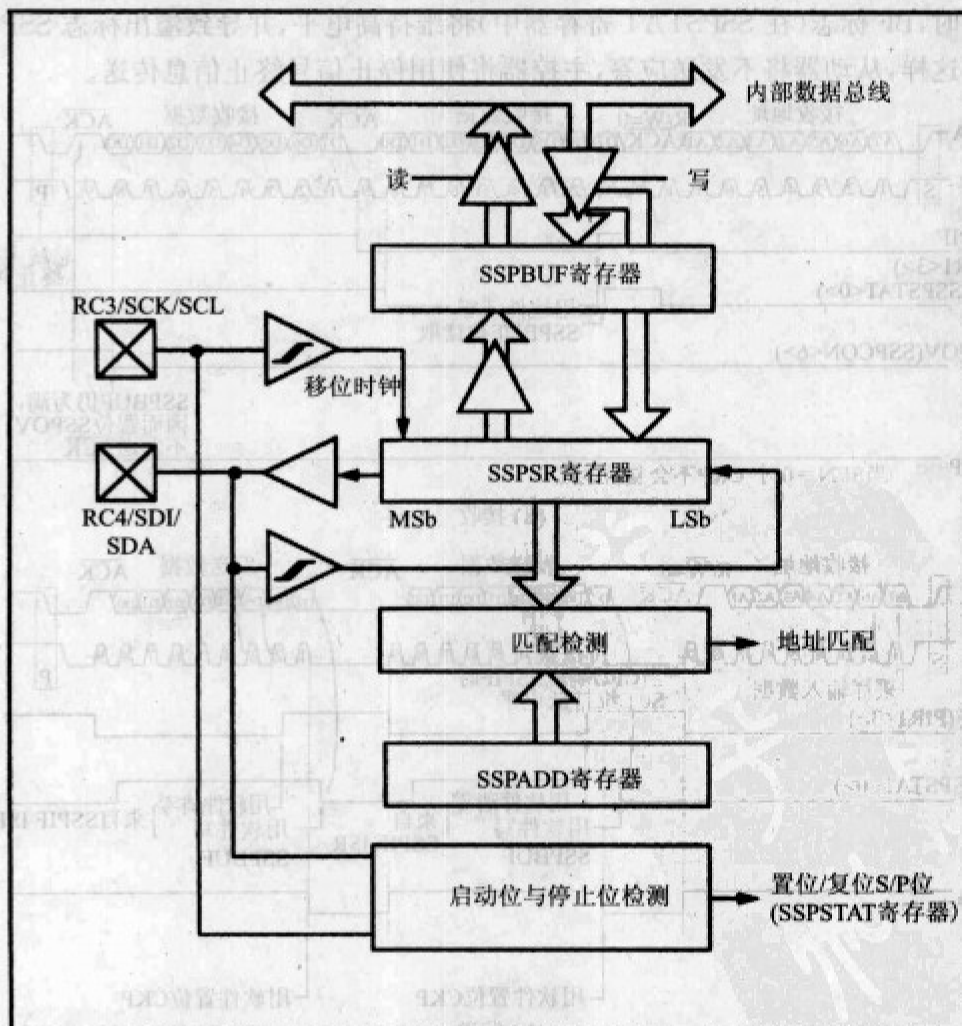


图 10-17 I²C从动模式框图

从动器所扮演的角色非常简单，当它被寻址时，就按照主控器的要求工作；未被寻址时就只是等待。因此，当它所具有的专用逻辑电路检测到启动信号发生时，地址匹配比较器可以识别出跟在启动信号之后的地址是否与其内部节点地址（保存在 SSPADD 控制寄存器中）相匹配。如果地址匹配，寄存器 PIR1 中的 MSSP 中断标志 SSPIF 将被置位。程序员可以利用此标志位来发起从动器响应。具体响应方式主要取决于地址字中 $R/\overline{W}$ 位的取值，它已被传送至 SSPSTAT 寄存器的 $R/\overline{W}$ 位中，可以在程序中进行检测。

如果程序检测出是一个写操作，从动器将作为接收器来工作。图 10-18a 给出了 7 位地址情况下的相关时序图。当接收到地址字节时缓冲满标志 BF 将被置位，因此需要读 SSPBUF 将其清零。在使用过 SSPIF 标志之后需要将其清零。然后，从动器将在主控器时钟控制下，随时钟接收 1 字节数据。如果一切运行正常，从动器将自动产生应答信号（Acknowledge）。图 10-18a 还给出了下列情况：当第 1 个字节未从 SSPBUF 中读取时，BF 标志（在 SSPSTAT 寄存器中）将维持高电平，并导致溢出标志 SSPOV 被置位。这样，从动器将不发送应答，主控器将使用停止信号终止信息传送。

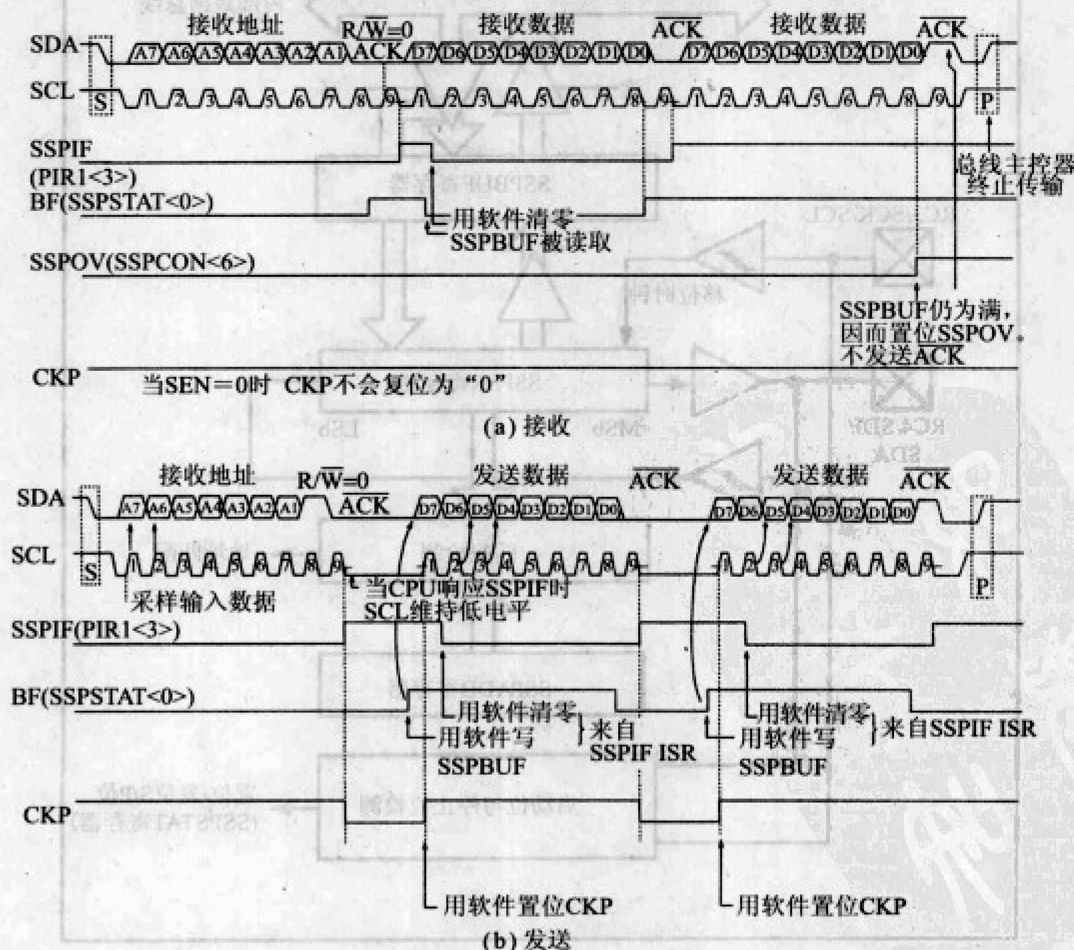


图 10-18 I^2C 从动模式时序（7 位地址，SEN＝0）

如果程序在信息的第1个字节中检测出读操作，从动器将作为发送器来工作。图10-18(b)显示了相关时序。当从动器识别出它已被寻址并且需要进行读操作时，它必须向SSPBUF寄存器写入1字节数据，此时BF标志将自动被置为高电平。当然，要作出响应需要一点时间，因此它需要将时钟线维持在低电平，直至完成SSPBUF的写操作。这可以阻止串行连接上的任何动作——从动器具有这种能力。在软件中将CKP置为高电平就可以释放串行线。在完成9位数据传输之后，硬件将自动将其置为低电平(当BF为低时)。这就是"时钟延长(clock stretching)"的一个例子，它是从动器对串行连接上的活动施加影响的几种方式之一。这种方式在发送模式下是自动实现的，而在接收模式下则是可选的，可以通过置位SSPCON2寄存器中的SEN位来实现。 281

当CKP被释放之后，主控器将随时钟接收数据，并产生应答信号作为响应。通过读取SSPSTAT状态寄存器中的各个状态位可以获得更多状态信息。当检测到主控器发出的停止信号之后，将终止I²C信息传输。认识了上述过程，就可以很好地理解图中给出的时序。 282

10.7.3 I²C主控模式下的MSSP

如图10-19所示，从整体上看，主控模式下的MSSP非常复杂。毕竟，主控器必须控制所有的总线交互。然而，模块核心依然是SSPSR移位寄存器和SSPBUF缓冲器。从图中还可以看到前面已经提到的I²C的特有连接，即2根串行线构成的电气接口，并具有开漏输出和施密特触发器输入。时钟信号是由内部的波特率发生器所产生的。 283

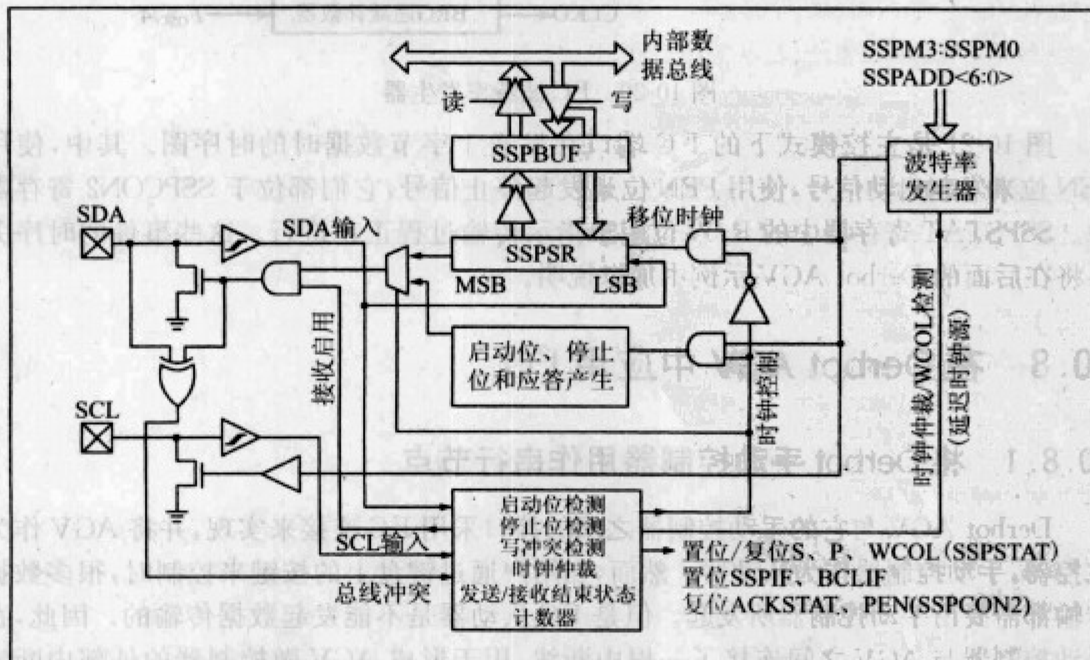


图10-19 I²C主控模式下的MSSP框图

它被送入移位寄存器，并通过 SCL 引脚送至外部总线。这类节点必须能够产生和检测启动信号、停止信号和应答信号。它还通过在 SDA 输出端上连接异或门，实现总线冲突检测功能。当主控器在发送数据时，如果总线上的逻辑状态与它所发送的值不一致，将产生总线冲突。总线冲突时将置位总线冲突中断标志 BCLIF（见图 7-10）。当主控器发现它正与其他主控器竞争时，主控器端口还可以实现仲裁功能。

图 10-19 右上角所示的波特率发生器的详细结构如图 10-20 所示。它具有一个递减计数器，当递减到 0 时，将重新装入 SSPADD 寄存器中保存的值。注意，当模块处于主控模式时，SSPADD 寄存器与地址无关，因为主控器没有地址。用户通过设置 SSPADD 的值选择所需的波特率。下式给出了相应的计算式，引自参考文献 10.3：

284

$$[SSPADD]=\frac{F_{OSC}}{4\times F_{SCL}}-1 \tag{10-1}$$

上式中，[SSPADD]是向 SSPADD 寄存器装入的值，F_{OSC} 是微控制器的时钟频率，F_{SCL} 是所需的 I²C 时钟频率。

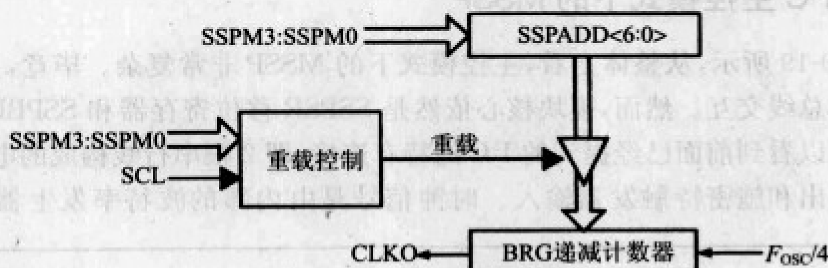


图 10-20 I²C 波特率发生器

图 10-21 是主控模式下的 I²C 端口在发送 1 字节数据时的时序图。其中，使用 SEN 位来发起启动信号，使用 PEN 位来发起停止信号，它们都位于 SSPCON2 寄存器中。SSPSTAT 寄存器中的 R/$\overline{W}$ 位用于指示传输过程正在进行。这些事件的时序关系将在后面的 Derbot AGV 示例中加以说明。

285

10.8 在 Derbot AGV 中应用 I²C

10.8.1 将 Derbot 手动控制器用作串行节点

Derbot AGV 与它的手动控制器之间的接口采用 I²C 连接来实现，并将 AGV 作为主控器，手动控制器作为从动器。然而，当用户通过键盘上的按键来控制时，很多数据传输都需要由手动控制器所发起。但是 I²C 从动器是不能发起数据传输的。因此，在手动控制器与 AGV 之间连接了一根中断线，用于形成 AGV 微控制器的外部中断输入。整个交互过程都是事先编程好的，这样当控制器检测到有键盘按下时，它将向 AGV 发送中断，然后 AGV 通过 I²C 连接发出数据请求。

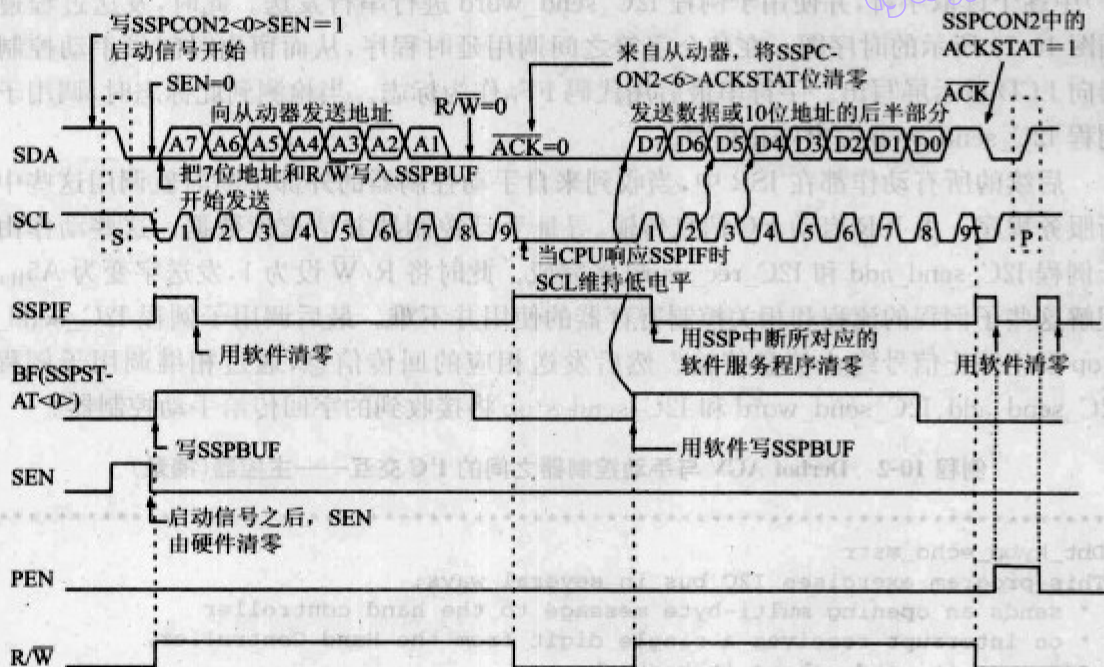


图 10-21 I²C 主控模式时序(发送,7 位或 10 位地址)

下面的 2 个例程 10-2 和例程 10-3 分别是为 Derbot AGV(作主控器)和手动控制器(作从动器)所编写的。当二者协同工作时,手动控制器上的按键动作将引发中断并传送至 AGV。之后这将形成一个 I²C 信息,由作为主控器的 AGV 向手动控制器请求 1 字节数据(或字符)。然后,AGV 将该字符传回控制器,并显在显示屏上。本书附属资源中包含有完整的程序。其中的子例程和 ISR 可以当作任何 I²C 器件之间的基本通信程序,可以用在手动控制器、AGV 以及用户所设计的任何其他 I²C 外围设备中。但需要注意的是,这些程序都相对简单,可能无法检测或响应某些故障信号。

10.8.2 将 AGV 用作 I²C 主控器

例程 10-2 在 Derbot AGV 上运行,并将其 I²C 端口用作主控器。一些关键的 SFR 设置包括端口 C(在这里必须将 I²C 端口位设置为输入)、SSPADD(在这里设置时钟频率)和 SSPCON1(在这里进行总体设置)。使用公式(10-1)可以确定 SSPADD 的值。SSPCON1 的详细设置可以参看图 10-14 中对寄存器的详细描述。

程序中所有主要的 I²C 动作都使用子例程来完成,认真查看这些程序会对我们有所帮助。程序由注释开始,然后发送起始字符串,向手动控制器发送字符信息"Derbot"。通过在一次信息发送中包含多个字节来完成。首先使用子例程 I2C_send_add 发送地址。地址 52_H 是随意选择的,必须将其左移 1 位放入地址字。将 $R/\overline{W}$ 设为 0,发送字变为 $A4_H$。

在这个正在发送的 I²C 信息中,程序从 Table 1(例程中未给出,但在本书附属资源

中）中逐个读取字符，并使用子例程 I2C_send_word 进行串行发送。此时，发送过程遵循图 10-21 所示的时序图。在各个字符之间调用延时程序，从而留出时间供手动控制器向 LCD 显示屏写出。字符串最后用代码 FF_H 作为标志。当检测到此标志时，调用子例程 I2C_send_stop 发出停止信号。

后续的所有动作都在 ISR 中，当收到来自手动控制器的外部中断时就调用这些中断服务程序。由 ISR 启动 I^2C 信息传输，寻址手动控制器并请求读数据。这些动作由 286 子例程 I2C_send_add 和 I2C_rec_word 来完成。此时将 $R/\overline{W}$ 设为 1，发送字变为 $A5_H$。理解这些子例程的流程和相关控制寄存器的使用并不难。最后调用子例程 I2C_send_stop 发出停止信号终止信息传输。然后发送相应的回传信息，通过相继调用子例程 I2C_send_add、I2C_send_word 和 I2C_send_stop 将接收到的字回传给手动控制器。

例程 10-2 Derbot AGV 与手动控制器之间的 I^2C 交互——主控器（摘录）

```
;**************************************************************************
;Dbt_kybd_echo_mstr
;This program exercises I2C bus in several ways:
; * sends an opening multi-byte message to the hand controller
; * on interrupt receives a single digit from the Hand Controller,
; * stores it, and echoes it back.
;Routines can be embedded into any program to provide user control of AGV.
;TJW 20.7.05                              Tested and working 21.7.05
;**************************************************************************
...
(opening program sections omitted)
...
;Specify RAM
I2C_PX_word     equu    23      ;holds most recent I2C word recd
I2C_add         equ     24      ;holds address used in I2C message
I2C_TX_word     equ     25      ;holds word to be transmitted on I2C
...
        org 00
        goto start
        org 04
        goto    Interrupt_SR
;Initialise SFRs in Bank 1
start   bcf     status,rp1
        bsf     status,rp0      ;select memory bank 1
...
        movlw   B'10011000'     ;set port C bits, I2C bits are both set as ip
        movwf   trisc
        movlw   07              ;set up 125kHz baud rate
        movwf   sspadd
;Initialise SFRs in Bank 0
        bcf     status,rp0
        movlw   B'00101000'     ;SSPCON1:MSSP on, I2C Master
        movwf   sspcon
...
;Send opening string
```

```
        clrf    pointer
        movlw   0a4              ;send slave address, R/W is write
        movwf   I2C_add
        call    I2C_send_add
loop_strl movf pointer,0
        call    table1
        movwf   I2C_TX_word
        sublw   0ff              ;test and move on if end marker reached
        btfsc   status,z
         oto    string_end
        call    I2C_send_word
        incf    pointer,1
        call    delay1           ;give LCD time to write
        call    delay1
        call    delay1
        goto    loop_strl
string_end      call I2C_send_stop
;Enable interrupts
        bcf     intcon,intf        ;clear pending interrupts
        bsf     intcon,inte        ;enable external interrupt
        bsf     intcon,gie
;Wait for interrupts from Hand Controller
loop    goto    loop
;
;*****************************************************************
;ISR. On external interrupt, SSP reads byte from Hand Controller,
;and echoes it back, ie two I2C messages.
;Received Byte stored in I2C_RX_word for further action.
;*****************************************************************
Interrupt_SR
        bsf     portc,6          ;diagnostic
;Start new I2C message, requesting word from slave.
        movlw   0a5              ;this is slave address, R/W is read
        movwf   I2C_add
        call    I2C_send_add
;now wait for byte to come in
        call    I2C_rec_word
        call    I2C_send_stop
        bcf     status,rp0
        call    delay20u
;Now echo byte - start new message
        movlw   0a4              ;this is slave address, R/W is write
        movwf   I2C_add
        call    I2C_send_add
;send the echoed character
        movf    I2C_RX_word,0 ;move received word to transmit store
        movwf   I2C_TX_word
        call    I2C_send_word
        call    I2C_send_stop
        bcf     status,rp0
        bcf     portc,6         ;clear diag led
```

```
        bcf     intcon,intf
        retfie
;*******************************************************************
;SUBROUTINES
;*******************************************************************
;initiates I2C message, by sending the word found in I2C_add, which
;must include R/W bit. Waits for all acknowledgement and completion
;states. Leaves RAM in Bank 0.
I2C_send_add
        bsf     status,rp0
        bsf     sspcon2,sen     ;force start bit
        btfsc   sspcon2,sen     ;check for its completion
        goto    $-1
        bcf     status,rp0
        movf    I2C_add,0       ;load address and data dirn bit
        movwf   sspbuf          ;and send
        bcf     pir1,sspif      ;will test this soon
        bsf     status,rp0
        btfsc   sspstat,bf      ;test for write complete
        goto    $-1
        btfsc   sspcon2,ackstat ;wait for 0 acknowledge bit
        goto    $-1
        bcf     status,rp0
        btfss   pir1,sspif      ;test for int flag to show completion
        goto    $-1
        bcf     pir1,sspif
        return
;
;Receives(single)word from I2C bus,and stores in I2C_RX_word. Returns Ack of 1,
;signalling this is last byte. Leaves RAM in Bank 0
I2C_rec_word bsf status,rp0
        bsf     sspcon2,rcen    ;set receive enable bit
        btfss   sspstat,bf      ;wait for buffer full
        goto    $-1
        bcf     status,rp0      ;read the data
        movf    sspbuf,0
        movwf   I2C_RX_word     ;store it for use somewhere
        bcf     pir1,sspif      ;preclear int flag, as we are about to use it
        bsf     status,rp0
        bsf     sspcon2,ackdt   ;set required acknowledge state, 1 as it's
                                        ;last byte
        bsf     sspcon2,acken   ;and enable it
        bcf     status,rp0
        btfss   pir1,sspif      ;use interrupt flag to test for end of ack
        goto    $-1
        bcf     status,rp0
        return
;
;Sends word on I2C bus, and awaits acknowledgement. Leaves RAM in Bank 0.
I2C_send_word bcf status,rp0
        movf    I2C_TX_word,0   ;get the word
```

288

```
        movwf   sspbuf ;this starts the transfer
        bsf     status,rp0
        btfsc   sspstat,r_w    ;test for write complete
        goto    $-1
        btfsc   sspcon2,ackstat ;check for 0 acknowledge bit
        goto    $-1
        bcf     status,rp0
        return
;
;Sends I2C stop bit, and awaits completion. Leaves RAM in Bank 0.
I2C_send_stop bsf status,rp0
        bsf     sspcon2,pen    ;force stop bit
        btfss   sspstat,p      ;test for stop bit completion
        goto    $-1
        bcf     status,rp0
        return
...
```

289

10.8.3 将手动控制器用作 I²C 从动器

例程 10-3 运行于 Derbot 手动控制器上，它对例程 8-1 中的程序 keypad_test 进行了扩展，添加了向 AGV 进行 I²C 传输的内容。首先应注意控制寄存器的用法，可以和主控模式相比较。通过 SSPCON1 将节点设置为从动器，此时 SSPADD 用于保存从动器地址。SSPADD 保存从动器地址 52_H，左移 1 位放置。同样地，必须将端口 C 的 I²C 位设置为输入。另外必须启用 2 个中断，即端口 B 的电平变化中断和 MSSP 中断，前者用于检测按键动作，后者用于提醒微控制器已接收到 I²C 信息中的首个字节。相应地需要设置中断控制位 GIE、PEIE、SSPIE 和 RBIE。

如果在接收的地址字节上检测到地址匹配，将发生 MSSP 中断。中断程序将首先确定中断源。如果是 I²C 中断，程序将首先检测 SSPSTAT 中的 $D/\overline{A}$ 位，从而确定刚接收的字节是地址还是数据。如果是地址，将对 SSPBUF 寄存器(其中保存有地址字节)执行读操作——只是清零 SSPSTAT 中的 BF 位。然后对 SSPSTAT 中的 $R/\overline{W}$ 位进行检测。如果主控器正在请求写操作，那么将退出 ISR。程序将等待下一次中断，中断再次到来时说明已收到所需的数据字节。如果主控器正在请求读操作，那么程序将留在 ISR 中，并转向标号 Send_I2C。具体时序遵循图 10-18b 所示。保存在 kpad_char 中的字节将被放入 SSPBUF，SSPCON1 中的 CKP 位被置为高电平以释放时钟信号，然后对中断标志位进行检测以确定传输是否完成。

如果测试 $D/\overline{A}$ 位时确定接收到的字节是数据，那么程序将转向标号 ISR1 继续执行。此时数据已经放置在 SSPBUF 中，然后读取该寄存器并使用 lcd_write 子例程将字符发送至显示屏。其中还调用了 dig_pntr_set 子例程，可以在本书附属资源中的完整程序清单中详细查看。该程序用于管理 LCD 指针，确保字符显示位置不超出所用的显示区。

例程 10-3 Derbot AGV 与手动控制器之间的 I²C 交互——从动器(摘录)

```
;************************************************************************
;dbt_kypd_echo_slave                          for Derbot Hand Controller
;Reads keypad value when pressed and sends interrupt to main AGV.
;Transmits on I2C keypad character when asked, and receives echo back.
;Displays anything sent from AGV.
;TJW 13.7.05                                             tested 15.7.05
;************************************************************************
...
(opening program sections omitted)
...
;Initialise SFRs in Bank 1
main    bcf     status,rp1
        bsf     status,rp0        ;select memory bank 1
...
        movlw B'00011000'        ;I2C bits of Port C to ip
        movwf trisc
        movlw B'10100100'
        movwf sspadd             ;our address to be 52H
                                 ;(it's shifted by one in sspadd)
        bsf    pie1,sspie         ;enable I2C interrupt
;
;Initialise SFRs in Bank 0
        bcf     status,rp0      ;select bank 0
        movlw   B'00110110'     ;SSPCON1:MSSP on, I2C Slave, 7 bit address,
                                ;interrupts off, no clock stretch on Receive
        movwf   sspcon
...
;enable global interrupts
        clrf    portb   ;initialise keypad value
        bsf     intcon,gie
        bsf     intcon,peie
loop    goto    loop            ;await keypad and I2C interrupts
;
;**************************************************************
;This is ISR, caused by keypad or I2C address match.
;Does not context save, as all action is in ISRs.
;**************************************************************
Interrupt_SR  btfsc intcon, rbif ;is it keypad interrupt?
        goto    kpad_ISR
;Here if interrupt is I2C, either address match (Ack sent automatically)
;OR further received byte has been detected.
;check whether this byte was address or data
        bsf     status,rp0
        btfsc   sspstat,d_a
        goto    ISR1            ;go if word was data
        bcf     status,rp0
        movf    sspbuf,0        ;dummy read of the address byte, to clear flag
;check if read, if so load and send byte
        bsf     status,rp0
        btfsc   sspstat,r_w
        goto    Send_I2C
```

```
       bcf     status,rp0      ;otherwise exit ISR, to await incoming data byte
       bcf     pir1,sspif      ;clear interrupt bit, and end ISR
       retfie
;Here if data byte has been detected, word is hence already in buffer.
ISR1 call dig_pntr_set         ;sort display pointer
       bcf     status,rp0      ;read word
       movf    sspbuf,0
       movwf   I2C_RX_word     ;save word
       movwf   lcd_op          ;prepare to send word to display
       bsf     portc,lcd_rs
       call    lcd_write
;transfer to lcd is done, end ISR.
       bcf     pir1,sspif      ;clear interrupt bit
       retfie
;here if sending I2C word. Send byte held in kpad_char
Send_I2C bcf   status,rp0
       bcf     pir1,sspif
       movf    kpad_char,0     ;move character to sspbuf
       movwf   sspbuf
       bsf     sspcon,ckp      ;release clock
       btfss   pir1,sspif      ;wait for completion of transfer
       goto    $-1
;transfer is complete, end ISR.
       bcf     pir1,sspif      ;clear interrupt bit
retfie
;
;Keypad press has been detected through Port B Interrupt on Change.
;Gets value, converts to character, stores in kpad_char, awaits key release,
;and sends interrupt to AGV
kpad_ISR    call    kpad_rd
...
```

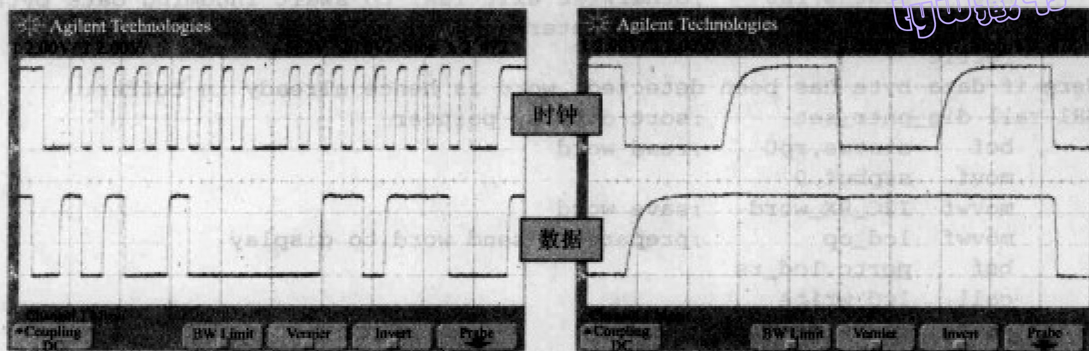
291

10.8.4 Derbot I²C 程序验证

图 10-22 显示出上述程序运行过程中 I²C 交互信号的部分波形。从图 10-22a 中可以看出，在 I²C 数据交换中，每个字对应 9 个时钟周期，这是一个很显著的特点。图中时钟频率略低于期望频率 125kHz。主控器通过发送地址字节 10100100_B(即 $A4_H$)启动信息传输。其中 LSB 是 $R/\overline{W}$ 位，表示请求写操作。从动器将以信息字 00110111_B(即 37_H，7 的 ASCII 码，表示键“7”被按下)作为回应。然后主控器做出应答，并施加停止信号。

图 10-22b 详细显示了开漏输出端对逻辑线的驱动特性。当晶体管输出打开时，信号线可以很快地从逻辑 1 跳变到逻辑 0。然而，当晶体管输出关闭时，信号线只是通过上拉电阻的动作逐渐上升到逻辑 1，并且从图中可以看出这是一个相对较慢的指数上升过程。

前面讨论的 2 个程序都很有用，但同时这些 I²C 程序仅限于某些特定的应用。尤其需要注意的是，这 2 个程序并没有考虑到 I²C 节点可能进入的所有状态，也没有考虑在串行连接工作不正常的情况下该怎样正确地工作。例如，在主控器程序中，当未接收到应答信号时，程序只是简单地无限循环。



(a) 完整的单字节信息 (b) 信号边沿的细节波形

图 10-22 I²C 的实际波形

292

10.9 对同步串行数据通信的评价及对异步通信方式的介绍

通过前面的讨论可以发现，同步串行通信是一种极为有用的数据传输方式，但存在这样一个问题：是否真有必要随数据流发送时钟信号？虽然使用时钟信号可以方便地实现数据同步，但这种方式存在下面几个缺点。

- □ 每个数据节点都需要连接一根额外的线路。
- □ 时钟所需带宽总是数据所需带宽的 2 倍；因此，对时钟信号的要求限制了总的数据传输速率。
- □ 对于远距离传输，时钟和数据均会失去同步特性。

10.9.1 异步原理

基于上述原因，目前已经开发出一些串行标准，它们不需要随数据发送时钟信号。这种通信方式通常称为异步(asynchronous)串行通信。此时，需要接收器从它所接收的信号中直接提取所有的时序信息。采用这种通信方式，需要对信号施加新的要求，同时，发送器和接收器节点也会比同步节点略显复杂。

一种用于解决异步通信所面临的这些挑战的方法(但不是唯一的方法，人们只要发挥聪明才智，就能够建立多种不同的异步连接)，通常要基于以下条件。

- □ 数据速率是预先确定的——发送器和接收器都经过预先设置从而识别出相同的数据速率。这样每个节点都需要精确稳定的时钟源，用来产生波特率。而对于实际值与理论值之间的微小偏差，是可以接受的。
- □ 每个字节或字都通过帧(frame)的方式来传输，包括 1 个起始位和 1 个停止位，用于实现数据传输之前的同步。

图 10-23 按顺序显示了某些标准(如 RS-232)所使用的异步数据格式。当线路空闲时将停留在一个预先设定的状态上。数据字的传输由起始(Start)位所发起，该位与空闲状态的极性相反。起始位的前跳边沿用作同步信号。然后，将随时钟接收 8 个数

据位。有时也使用第 9 位作为奇偶校验位。之后线路返回空闲状态,形成停止(Stop)位。在单个停止位完成之后,可以立刻发送新的数据字,否则线路将保持空闲状态直至下次传输。

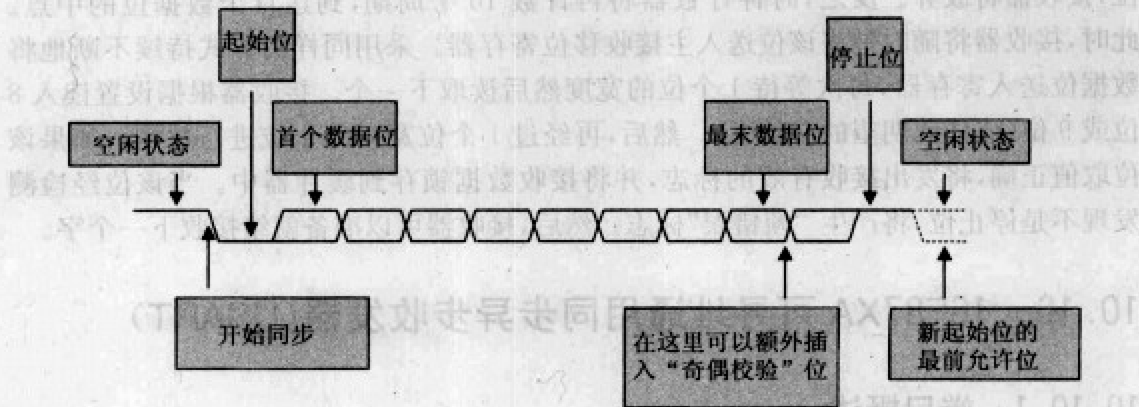


图 10-23 异步串行数据的一般格式

10.9.2 在不接收时钟信号时如何对串行数据进行同步

为了在没有伴随时钟信号的情况下正确地接收外来数据,接收器必须能够检测到字节或字的起始位置,以及每个数据位中数据保持有效的时刻。由于数据速率是预先确定的,因此也许有人会认为这并非难事。但是对于不同的微控制器,要让它们的时钟频率保持精确一致是不可能的。

293

图 10-24 显示出这种时序原理是如何实现的。接收器有一个内部时钟,其运行频率正好是期望位速率的整数倍。通常选择 16 倍,但这并不是必须的。接收器对串行

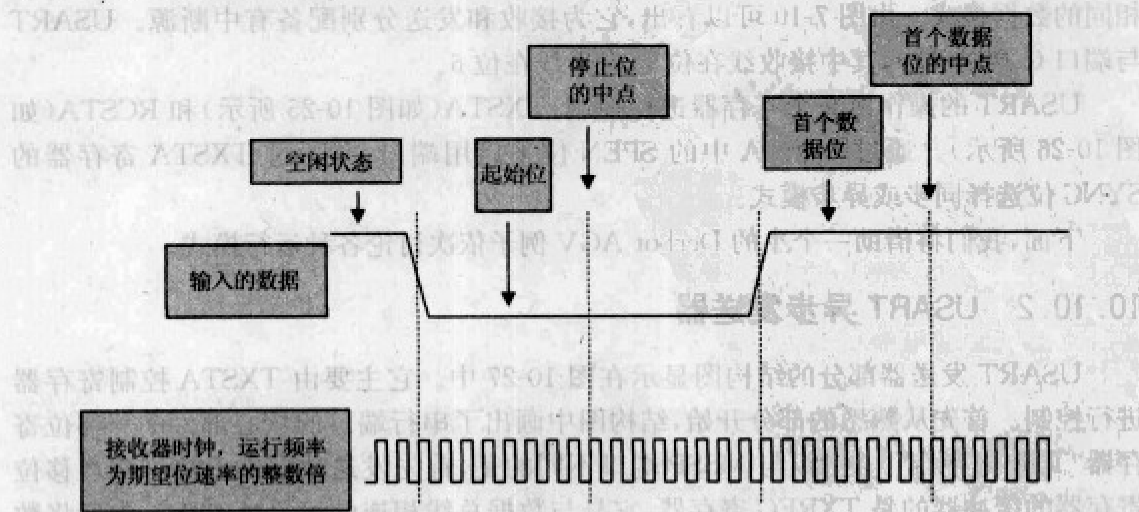


图 10-24 对异步数据信号进行同步

接收线上的外来数据状态进行监视。当检测到起始位时，计数器开始对时钟周期进行计数，直至到达期望起始位的中点。也即当使用×16 时钟时，中点在第 8 个时钟周期。此时，它将再次检测外来数据线的状态，从而确定是起始位。如果检测发现不是起始位，接收器将放弃。反之，时钟计数器将再计数 16 个周期，到达首个数据位的中点。此时，接收器将随时钟将该位送入主接收移位寄存器。采用同样的方式持续不断地将数据位送入寄存器，每次等待 1 个位的宽度然后读取下一个。接收器根据设置读入 8 位或 9 位(或任何期望的字长度)。然后，再经过 1 个位宽对停止位进行检测。如果该位取值正确，将发出接收有效的标志，并将接收数据锁存到缓冲器中。当该位经检测发现不是停止位，将产生“帧错误”标志。然后，接收器可以准备就绪接收下一个字。

294

10.10 16F87XA 可寻址通用同步异步收发器(USART)

10.10.1 端口概述

16F87XA 型号所具有的另一种串行端口是可寻址通用同步异步收发器(Addressable Universal Synchronous Asynchronous Receiver Transmitter，USART)。这个相当拗口的名称表明，它可以运行于同步或异步模式下，且既可以接收又可以发送。名称中包含的字符“通用”反映出它的传统叫法，直接表明它可以配置为所需的所有主要的运行模式。“可寻址”是一种使用方式，可以选定某个输入字节并将其翻译成地址。

可以将 USART 配置为同步主控器、同步从动器或异步模式。在后一种模式下，可以实现全双工通信，也即它可以同时接收和发送。这样，它就同时拥有接收移位寄存器和发送移位寄存器，二者可以同时运行。这两部分共享同一波特率发生器并且具有相同的数据格式。由图 7-10 可以看出，它为接收和发送分别配备有中断源。USART 与端口 C 共用引脚，其中接收线在位 7，发送线在位 6。

USART 的操作由 2 个寄存器进行控制：TXSTA(如图 10-25 所示)和 RCSTA(如图 10-26 所示)。通过 RCSTA 中的 SPEN 位来启用端口，并通过 TXSTA 寄存器的 SYNC 位选择同步或异步模式。

下面，我们将借助一个小的 Derbot AGV 例子依次讨论各种运行模式。

10.10.2 USART 异步发送器

USART 发送器部分的结构图显示在图 10-27 中。它主要由 TXSTA 控制寄存器进行控制。首先从熟悉的部分开始，结构图中画出了串行端口的核心部件——移位寄存器“TSR 寄存器”。注意，与 MSSP 端口不同的是，首先发送数据的 LSB。充当移位寄存器的缓冲器的是 TXREG 寄存器，它是与数据总线相连的可寻址 SFR。程序将数据写入该寄存器。移位寄存器由“波特率 CLK”时钟驱动，该时钟由 TXEN 位来启用。时钟频率由波特率发生器控制，取决于 SPBRG 寄存器中保存的值。移位寄存器的输

出通过“引脚缓冲与控制”电路连向微控制器引脚。该电路由 RCSTA 控制寄存器中的串行端口使能(Serial Port Enable)位(SPEN 位)启用。

R/W-0	R/W-0	R/W-0	R/W-0	U-0	R/W-0	R-1	R/W-0
CSRC	TX9	TXEN	SYNC	—	BRGH	TRMT	TX9D

位 7　　　　　　　　　　　　　　　　　　　　　　　　　位 0

位 7　**CSRC**：时钟源选择位
异步模式
无关
同步模式
1＝主控模式（时钟由BRG内部产生）
0＝从动模式（时钟来自外部时钟源）

位 6　**TX9**：9位发送使能位
1＝选择9位发送
0＝选择8位发送

位 5　**TXEN**：发送使能位
1＝发送启用
0＝发送禁止
注：在同步模式下，SREN/CREN比TXEN优先级高

位 4　**SYNC**：USART模式选择位
1＝同步模式
0＝异步模式

位 3　**未实现**：读作“0”

位 2　**BRGH**：高速波特率选择位
异步模式
1＝高速
0＝低速
同步模式
此模式下未用

位 1　**TRMT**：发送移位寄存器状态位
1＝TSR空
0＝TSR满

位 0　**TX9D**：发送数据的第9位，可以作为奇偶校验位

图 10-25　发送状态与控制寄存器 TXSTA(地址 98_H)

将要发送的数据必须首先由程序装入 TXREG 寄存器。如果此时没有发送动作，或者发送已经开始但已过了停止位，那么 TXREG 中的数据将立刻被送入 TSR 移位寄存器。状态信息通过 2 个位来提供：中断标志位 TXIF 和 TRMT 位。前者用于指示 TXREG 状态。当 TXREG 和移位寄存器之间发生数据传输时，中断标志 TXIF(在寄存器 PIR1 中，见图 7-12)将被置位。该位无法用软件清零，只有当 TXREG 被重新装载时才能清零。TRMT 位用于监测移位寄存器的状态，可以由程序进行轮询。当移位寄存器为空(即发送已结束)时，该位被置位。

在所发送的字中可以插入第 9 个数据位 TX9D，它由 TX9 位启用。这 2 个位都位于 TXSTA 控制寄存器中。第 9 个数据位可以用作奇偶校验位。但是，与某些串行端口不同的是，奇偶校验值并不是由硬件自动产生的，而是由程序员在程序中完成的。如果使用了第 9 位，那么在相关数据字写入 TXREG 之前就应当建立该位。如果事先没有设定好该位，那么向 TXREG 写入数据将会在第 9 位就绪之前启动串行数据传输。第 9 位的另一个用途是指示出紧跟在后面的字节是一个地址，详见 10.10.6 节。

295
~
296

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R-0	R-0	R-X
SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D
位 7							位 0

位 7　**SPEN**：串行端口使能位
　1＝串行端口启用（将RC7/RX/DT和RC6/TX/CK引脚配置为串行端口引脚）
　0＝串行端口禁止

位 6　**RX9**：9位接收使能位
　1＝选择9位接收
　0＝选择8位接收

位 5　**SREN**：单字节接收使能位
　异步模式
　无关
　同步主控模式
　1＝启用单字节接收
　0＝禁止单字节接收
　该位在接收完成后被清零
　同步从动模式
　无关

位 4　**CREN**：连续接收使能位
　异步模式
　1＝启用连续接收
　0＝禁止连续接收
　同步模式
　1＝启用连续接收，直至使能位CREN位被清零（CREN优先级高于SREN）
　0＝禁止连续接收

位 3　**ADDEN**：地址检测使能位
　9位异步模式（RX9＝1）
　1＝启用地址检测，当置位RSR<8>时允许中断及装入接收缓冲器
　0＝禁止地址检测，接收所有字节，第9位可用作奇偶校验位

位 2　**FERR**：帧出错标志位
　1＝帧出错（读RCREG寄存器可更新该位并接收下一个有效字节）
　0＝无帧错误

位 1　**OERR**：溢出错误位
　1＝溢出错误（清零CREN位可将此位清零）
　0＝无溢出错误

位 0　**RX9D**：接收数据的第9位（可用作奇偶校验位，但必须由用户固件计算得到）

图 10-26　接收状态与控制寄存器 RCSTA（地址 18_H）

297

10.10.3　USART 波特率发生器

在 USART 的同步模式和异步模式下，都用到了波特率发生器。它的核心是一个自由运行的 8 位计数器，受控于 SPBRG 寄存器。该计数器时钟来自微控制器内部的振荡器频率，计数器动作的效果就是将内部振荡器频率除以某个数，该除数取决于 SPBRG中的数值。在异步模式下，还将 TXSTA 寄存器中的 BRGH 作为预设比例因子对该除数进行修正。最终得到的波特率频率如下，其中[SPBRG]是同名寄存器的值：

注：TXIF、TXIE参见图7-12。除SPEN之外，其他所有控制位在TXSTA(图10-25)中

图 10-27 USART 发送框图(阴影框中所附标签为作者所加)

异步模式下：

$$\text{当 } \mathbf{BRGH}=0 \text{ 时，波特率}=\frac{f_{OSC}}{64([\mathbf{SPBRG}]+1)} \tag{10-2}$$

$$\text{当 } \mathbf{BRGH}=1 \text{ 时，波特率}=\frac{f_{OSC}}{16([\mathbf{SPBRG}]+1)} \tag{10-3}$$

同步模式下：

$$\mathbf{BRGH}=\text{无关，波特率}=\frac{f_{OSC}}{4([\mathbf{SPBRG}]+1)} \tag{10-4}$$

10.10.4 USART 异步接收器

USART 接收器部分的结构图如图 10-28 所示。它主要受控于 RCSTA 寄存器(如图 10-26 所示)。在某些方面，可以将这张图看作图 10-27 的发送结构图的镜像。但由于接收器与发送器具有不同的要求，因此二者在某些特性上又完全不同。数据由微控制器的端口 C 的位 7 输入。其中还加入了数据恢复(data recovery)电路以降低干扰。每确定 1 个数据位，都需要采样 3 次，并取其中出现次数较多的数值进行传输。位于接收器中央的是移位寄存器“RSR 寄存器”。它由来自波特率发生器的时钟所驱动。在移位寄存器与数据总线之间通过 FIFO(先进先出)缓冲器实现双缓冲。该 FIFO 缓冲器的高字节为 RCREG 寄存器，是可寻址的 SFR。这种双缓冲器可以保存最近已接收到的 2 个字节，从而允许第 3 个输入字节随时钟进入移位寄存器。端口通过 SPEN 位进行全局使能。优先级低于 SPEN 的是单字节接收使能位 SREN(Single Receive ENable，请勿与 SPEN 混淆)或连续接收使能位 CREN(Continuous Receive ENable)。通过它们可以选择接收单个输入字或接收一个连续序列。

当接收到包括停止位在内的完整字之后，如果 FIFO 有空间供接收，那么该字中主要的 8 位数据将被传送至 FIFO，并且中断标志位 RCIF 被置位。如果使用了第 9 位(取决于 RX9 位的状态)，那么该位将被传送至 RX9D。该位与主要输入字节具有相同

的缓冲深度。

注：1. FIFO，First In First Out，先进先出
2. RCIF、RCIE，参见图7-6RCIF、RCIE，参见图7-12
3. 其他所有控制位在RCSTA（图10-26）中

图 10-28 USART 接收框图（阴影框中所附标签为作者所加）

299

如果已随时钟输入 1 个数据字节，但检测到停止位为逻辑 0，那么将通过 FERR 位指示出帧错误。与 RCREG 和 RX9D 类似，该位也是双缓冲的，这样就可以检测出每个缓冲位的帧状态。如果 RCREG 的 2 个缓冲区都为满，同时 RSR 寄存器又接收到 1 个完整字节，那么将置位溢出错误位 OERR。RSR 寄存器中的字将丢弃，并且从 RSR 寄存器向 RCREG 的数据传输将被阻止。在继续接收数据之前，必须通过软件将 OERR 标志清零。当读取数据值时，对 RCREG 的读取动作将更新 FERR 和 RX9D 的值。因此，在读 RCREG 之前应先读取 FERR 和 RX9D 这些特殊的位值。

10.10.5 异步通信示例

本节将在前面讲述的例程 10-2 的基础上进行扩展，添加异步通信连接的内容。现在，当 Derbot 手动控制器上的按键按下时，按键字符被发送至 AGV，然后 AGV 将字符送到它的异步发送器上。通过异步通信，字符被回传给异步接收器，并最终显示在手动控制器上。虽然这个额外的串行连接并没有提供任何附加功能，但却提供了很好的机会，可以开发简单的测试程序并观察更多的串行数据。例程 10-4 显示了与异步通信有关的细节。完整的程序在本书附属资源中。

通过阅读程序注释，并与图 10-25 和图 10-26 所示的控制寄存器进行比较，可以方便地理解 TXSTA 和 RCSTA 寄存器的初始设置。波特率 50 是随意选择的。

与例程 10-2 类似，所有的数据传输都发生在主 ISR 中。因此，没有使用异步中断。

ISR 首先通过 I²C 端口从手动控制器读取 1 个字节的数据，然后存放在 I2C_RX_word 中。例程给出了异步发送器发送数据的程序段。刚才接收到的字被送入 TXREG 寄存器中。在数据发送出去时，异步接收器将同时开始随时钟输入。程序只是简单地等待接收中断标志被置位，中断标志被置位表明已接收到该字并且传输结束。然后，通过 I²C 连接将接收到的字回传给手动控制器。

例程 10-4 Derbot 手动控制器上的异步数据传输

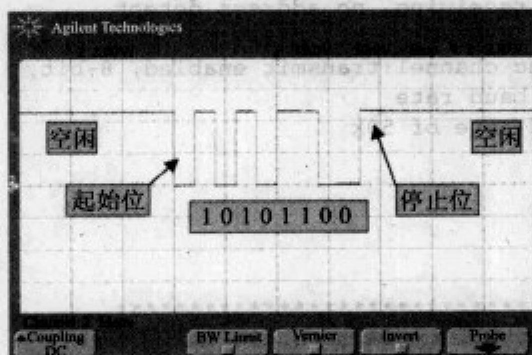
```
;****************************************************************************
;Dbt_kybd_echo_async
;This program receives a digit from the Hand Controller on the I2C
;bus,stores it, sends it through the asynchronous serial link,
;and echoes it back to the I2C. Each I2C message one byte only.
;Routines can be adapted and embedded into any Derbot program.
;TJW 18.7.05                              Tested and working 19.7.05
;****************************************************************************
...
(early program sections omitted)
...
       bcf       status,rp0
;Initialise USART in both banks
       movlw  B'10010000'   ;set up async channel: port is on, 8-bit transfer,
       movwf  rcsta         ;continuous receiving, no address detect
       bsf    status,rp0
       movlw  B'00100100'   ;set up async channel:transmit enabled, 8-bit,
       movwf  txsta         ;high speed baud rate
       movlw  04            ;set up baud rate of 50k
       movwf  spbrg
       bcf    status,rp0
...
(program sections omitted)
...
;*********************************************************************
;ISR. On external interrupt, SSP reads byte from Hand Controller,
;sends it out on USART, receives it back through USART
;and echoes it back to keypad.
;Received Byte stored in I2C_RX_word for further action.
;*********************************************************************
Interrupt_SR
...
;send out via async comm channel
       bcf    pir1,rcif      ;preclear receive interrupt flag
       movf   I2C_RX_word,0   ;get word, and move to txreg
       movwf  txreg
       btfss  pir1,rcif      ;test for receive interrupt flag,
                              ;indicating receive complete
       goto   $-1
       movf   rcreg,0        ;get and store received word
       movwf  async_RX_word
```

```
...
;send the echoed character
    movf    async_RX_word ;move async received word to transmit store
    movwf   I2C_TX_word
    call    I2C_send_word
...
```

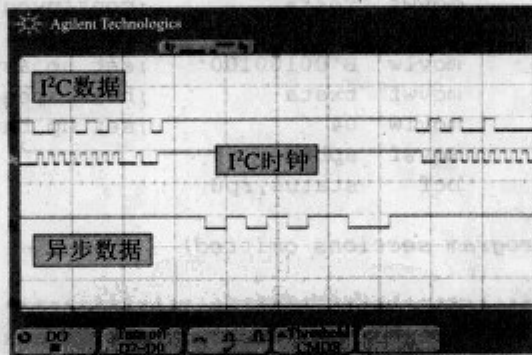
图 10-29 给出了例程产生的波形。图 10-29a 显示了 1 个字节的异步数据，以示波器的形式给出。水平时间基准为 50 μs/每格。可以看出，每位持续时间为 20 μs，正好对应于所选波特率 50。图中还可以清晰地看出，空闲状态为逻辑 1，最前面的起始位为逻辑 0。数据以“倒序”传输，即 LSB 在前。因此图中数据为 00110101_B，即 16 进制的 35_H。这是 ASCII 码 5，表示按下了该键。停止位已融入到下一个空闲时段中，是前面的 20 μs，因此不太容易看清楚。

图 10-29b 给出了同一个异步传输字，但此时它和前面及后面的部分 I^2C 信息显示在一起。此时，示波器处于逻辑分析器模式。前面的 I^2C 信息以它自己的格式显示出同样的数据字节 00110101_B，该字节由异步连接重复传输。后面的 I^2C 信息是手动控制器的地址，读作 10100100_B。其中 R/$\overline{W}$ 位为低电平(即主控器将对从动器执行写操作)，在第 9 个时钟周期上可以看到应答信号。

301



(a) 单字节异步通信



(b) 单字节由I²C传递到异步通信

图 10-29 串行波形

10.10.6 在 USART 接收模式下使用地址检测

在使用 USART 时，可以将地址嵌入到接收数据中。这样，就可以在串行线路上连接多个节点，每个节点可以识别出各自的地址。此时，必须将 USART 接收器设置为 9 位模式(位 RX9=1)，且 RCSTA 中的地址使能位 ADDEN 必须置为 1。一旦使用这种模式，第 9 位如果为逻辑 1，则表明紧随其后的字节是一个地址。传输开始时，第 9 位为 0 的所有数据都将被忽略。当检测到字节第 9 位为 1 时，程序将读取后续字节并(在软件中)确定该字节是否与自己的地址相匹配。如果地址匹配，程序需要清零 ADDEN，从而将接口恢复到正常的接收状态，后续字节将作为数据进行读取。这一过程持续进行直至检测到下一个地址字，而该地址可能属于其他节点。

10.10.7 USART 的同步模式

USART 主要用于实现异步功能,除此之外,通过设置 TXSTA 寄存器中的 SYNC 位可以选择 USART 的同步模式。此时必须通过 RCSTA 中的 SPEN 位启用端口。端口 C 中的位 7 用作串行数据线,位 6 用作串行时钟线。在本章早先介绍过的 MSSP 端口的 SPI 模式的基础上,要理解这种模式下的 USART 运行原理并不太难。因此,如果要使用 USART 的同步模式,可以自行查阅相关数据。

302

10.11 不借助串行端口实现串行通信——“bit banging”

从前面的叙述中似乎可以看出,要使用串行通信,就必须要使用带有 1 个或多个串行端口的微控制器。然而这并不正确,使用标准的输入/输出端口位,仅通过编程也可以产生或接收串行数据流。这是一种相对简单、同时又具有吸引力(从节省成本的角度看)的简单同步连接实现方式,尤其适用于只需偶尔使用串行连接的情况。其他更高级的协议(如 I^2C)也可以使用这种方式来实现,只是会更加困难。参考文献 1.1 中的第 6 章提供了更多信息,并给出了在 PIC 16F84 微控制器上的应用实例。

10.12 构建 Derbot 手动控制器

要运行本章使用的大部分程序,需要有一个具备工作 LCD 的手动控制器。另外,还需要合理连接 AGV 上的“总线”连接器以及 I^2C 上拉电阻。此时的电路就与图 A3-1 非常接近,只是没有光敏电阻和超声探测器。

小结

- ❑ 在嵌入式系统中,串行通信已变得日渐重要。优秀的设计员有必要对其加深理解。
- ❑ 串行通信主要分为 2 类:同步串行通信和异步串行通信。
- ❑ 串行通信有很多不同的标准与协议,其范围从最简单的到相当复杂的。重要的是对不同应用选择合适的协议。
- ❑ 16F873A 微控制器有 2 类极为灵活的串行端口。同时,灵活性的另一面是结构的复杂性,这导致掌握起来相当困难。因此,直接采用一些已有的程序,从而避免从头编写新代码是非常有必要的。

参考文献

10.1. The I²C Bus Specification, Version 2.1 (2000). Philips Semiconductors, 9398 393 40011.
10.2. I²C Manual (2003). Philips Semiconductors, AN10216-01.
10.3. Using the PICmicro® MSSP Module for Master I²C™ Communications (2000). Microchip Technology, AN735, DS00735A.
10.4. Using the PICmicro® SSP Module for Slave I²C™ Communications (2000). Microchip Technology, AN734, DS00734A.
303
10.5. Asynchronous Communications with the PICmicro® USART (2003). Microchip Technology, AN774, DS00774A.

第11章
数据采集与处理

本书前几章所讨论的内容都局限在数字信号的范围内。虽然数字信号为我们带来了很多便利，但要认识到，绝大多数的真实变量本质上都是模拟量。无论我们现在所讨论的是温度、音量、频率还是其他变量，它们都是连续变量，并且具有无数个不同的取值。因此，微控制器必须具有读取模拟量的能力，并且能够在必要时产生模拟量输出——虽然从内部结构来讲，微控制器绝对是一种数字器件。将模拟量转化为数字量的过程，以及后续的信号处理过程，通常称为“数据采集（data acquisition）”。

一旦采集到外部数据，就需要对其进行处理并加以利用。另外，还需要进行平均化、比例缩放、线性化以及存储。大多数情况下，这些数据将用于实现某种形式的控制目的，可能是进行显示，或者是发送至其他设备。

数据采集以及对所采数据的使用是本章的主要内容。本章将涉及以下主题：

- ❑ 数据采集系统的主要特性；
- ❑ 模数转换器的特性；
- ❑ 16F873A模数转换器的特性；
- ❑ 16F873A模数转换器的使用方法；
- ❑ 一些简单的数据处理技术；
- ❑ 比较器的使用以及16F873A比较器的性能。

一旦掌握了采集并处理数据的简单方法，我们就完全有能力制作一些测量设备。因此，本章最后将给出一些演示项目。其中，某些项目将Derbot用作AGV，而另外一些项目则只是将Derbot的核心设计用作其他项目的基础，而在这类项目中是不需要轮子的。

11.1 模拟量和数字量的采集与使用概述

多数传感器所产生的输出信号都是对它们所代表的某个量的模拟（analog）。因此，温度传感器的输出电压尽可能准确地表达它所代表的温度，并随着温度的升高或降低而变化。类似地，话筒的输出信号也尽可能准确地表达声波的相关特性，包括振幅、频率和波形。模拟信号具有较好的特性，但同时又受到很多严重缺陷的影响，如表11-1所示。比较而言，数字（digital）信号在很多方面都表现出更好的特性，并且在今天的技术环境下也更易于处理。这种比较优势在很多情况下是非常明显而且具有压倒性的。 304

使用模数转换器(analog-to-digital converter,ADC)可以非常方便地将信号从模拟方式转换为数字方式。用于实现这种转换的电路非常复杂。然而,相关设计已成为一项成熟的技术,可以直接使用现有的集成电路或微控制器内部的模块来实现。

作为嵌入式设计员,我们需要理解ADC的相关特性,从而选择合适的器件并有效地加以利用。

表 11-1 模拟量与数字量的某些特性

特 性	模 拟	数 字
(电气)表达方式	变量以连续可变的电压或电流来表示	变量以二进制数字表示
表达精度	可具有无数个取值。只要信号保持完全纯正,理论上可以实现绝对精度	只能以有限个数的数位组合来表示测量值;例如,8位数字只有256个不同组合。无法复现"连续可变"的模拟信号量
抵御信号降格的能力	几乎无法避免信号的漂移、衰减、失真和干扰。无法从这些降格状态下完全恢复出原信号	数字表达方式可以从本质上抵御多种形式的信号降格。还可以引入错误检测机制,使用合适的技术能够对降格信号进行完全恢复
处理能力	使用放大器和其他电路的模拟信号处理技术已经非常成熟,但是从根本上受限于灵活性,并总是存在信号降格的问题	可以使用非常强大的基于计算机的技术进行数字信号的处理
存储能力	几乎无法实现任意时长的真实模拟存储	所有主要的半导体存储技术都是以数字方式实现的

11.2 数据采集系统

在实现模拟信号向数字信号的转换过程中,仅仅选择合适的ADC通常是不够的。一般需要多个输入,并且需要在转换之前对信号进行处理。因此,在很多情况下都需要建立完整的数据采集系统(data acquisition system)。图11-1显示了这类系统的组成部分。图中以框图的方式显示出系统所具有的元素:多个输入、放大、滤波、信号源选择、采样与保持,以及最后的ADC。下面的几节将对不同元素进行概略的描述。

305

11.2.1 模数转换器

ADC的任务就是确定与输入电压等价的数字输出值。这类电路的设计是一项庞杂的工作。目前已经针对不同应用开发出很多差别很大的ADC电路。某些ADC(如双斜坡ADC)转换速度慢但却具有非常高的精度,适用于数字电压计等精确测量的场合。另外一些ADC(如Flash converter,即闪速转换器——勿与闪存技术相混淆)转换速度快,但精度较差,适用于转换视频或雷达等高速信号。还有一些ADC(如逐次逼近

ADC)的转换速度和精度介于二者之间,适用于多种目的的工业应用,它们是嵌入式系统中使用最多的 ADC 类型。在多数电子教材中都可以找到此类 ADC 电路的工作原理(见参考文献 1.1)。

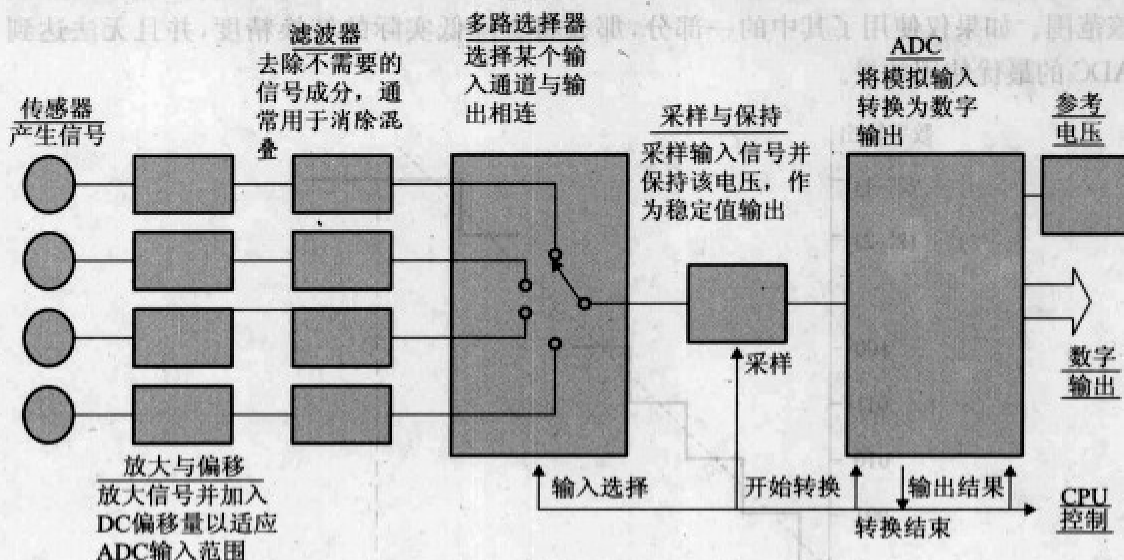


图 11-1 (4 通道)数据采集系统的组成元素

ADC 主要具有以下几个特性。

1. 转换特性

ADC 可以接受连续变化的输入电压,并将其转化为固定个数的输出值。相应的 ADC 转换特性示例如图 11-2 所示。图中,水平轴代表输入电压,垂直轴代表数字输出。如果 ADC 进行连续转换,输入电压从 0 开始逐渐增大,那么输出信号初始值也为 0。当输入到达某个值时,输出将变为…001。当输入进一步增大时,输出依然保持该值不变,直至输入增大到另一个特定值,这时输出就切换为…010。如果输入电压连续增大,输出将在某一点处到达最大值。此时,输入信号就贯穿了整个输入范围。而输出则以步进方式到达最大值。对于 n 位 ADC,最大输出值为(2^n-1)。例如,8 位 ADC 的最大输出值为(2^8-1),即 11111111_B或 255_D。

306

图 11-2 所示的输入范围从 0 开始,并上升至 V_{max}。V_{max}的放置位置比我们预想的稍向右偏,最后一格的中心似乎应该对应产生一个输出台阶 2^n。采用这种布局方式可以将水平轴精确地划分为 2^n个等长的区间,每个区间的中心对应一个输出跳变。

很多 ADC 都具有与图 11-2 类似的特性,例如输入范围为 0V~5V。然而,另外一些 ADC 则具有双极性输入范围,输入电压既可以取正值也可以取负值,例如-5V~5V。在所有情况下,输入范围 V_r都是指最大输入电压与最小输入电压的差。输入范围通常与参考电压(ADC 的一部分)有直接关系。

在图中可以通过直觉发现:输出位数越多,输出台阶数越多,转换效果越好。转换效果的度量标准称为精度(resolution)。它对应于当输出从一个数值上升至下一个数

值时,在输入上需要发生的变化量。在转换特性图中,精度对应于一个台阶的宽度。具有 n 个输出位的 ADC 可具有 2^n 个可能的输出值,从 0 直到(2^n-1)。因此,它的精度为 $V_r/2^n$,其中 V_r 为输入电压范围。输入信号应使用尽可能多的输入范围,但不要超出该范围。如果仅使用了其中的一部分,那么将会降低实际的转换精度,并且无法达到 ADC 的最优使用效果。

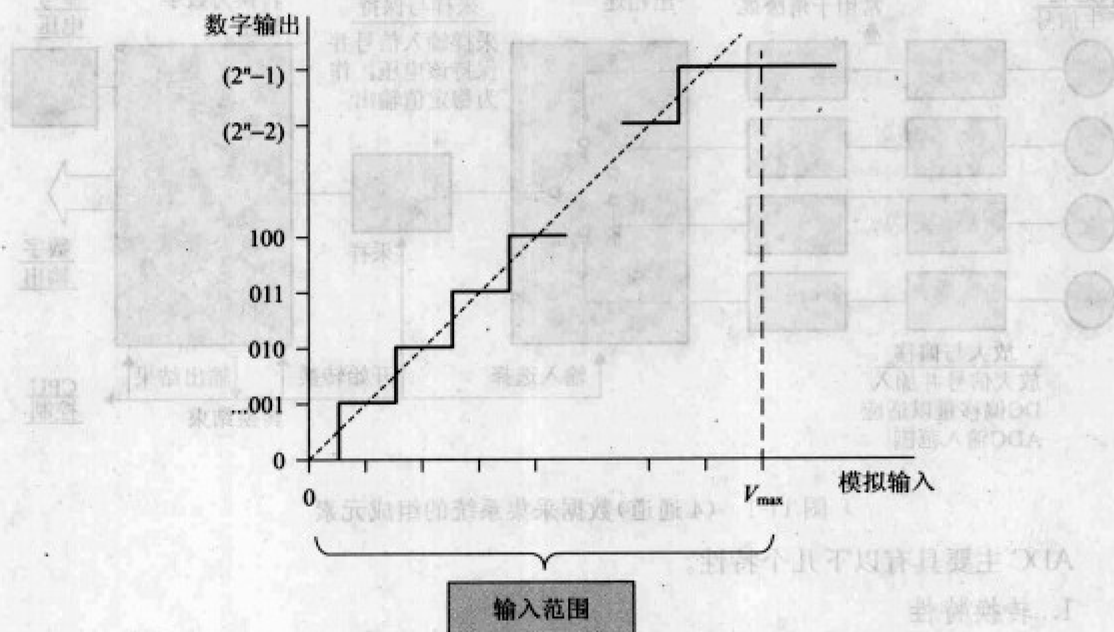


图 11-2 ADC 的理想输入/输出特性

2. 转换速度

ADC 完成转换所需的时间称为转换时间。转换时间较长的慢速 ADC 只能够转换低频信号,因为必须满足奈奎斯特准则(见 11.2.2 节)。ADC 的转换时间用于定义 ADC 可以转换的信号类型。前面已经提到,高精度的 ADC 完成转换通常需要较长的时间。

3. 数字接口

数字接口由控制信号和数据输出构成。图 11-1 给出了典型的控制信号。通常有一个信号用于启动 ADC 转换。转换完成时,ADC 将通过某个输出信号指示转换完成。另外还需要一个信号使 ADC 将转换后的数据输出。根据对接口类型的不同要求,ADC 具有并行或串行数据接口。

307

ADC 在工作时总是离不开参考电压(voltage reference)。这是一种能够维持非常精确稳定的电压的器件或电路,主要基于齐纳二极管或带隙参考电压。ADC 使用参考电压作为标准,对输入电压进行测量。ADC 的转换精度受限于参考电压的精度。要实现精确的 A/D 转换,应采用好的 ADC 器件并配合好的参考电压。

11.2.2 信号调理——放大与滤波

要达到ADC的最佳使用效果，输入电压在不超出范围的前提下，必须尽可能多地占满输入范围。然而对于多数信号源（如话筒或热电偶）而言，它们只能产生很小的电压。因此，多数情况下需要使用放大电路来达到输入范围的最佳效果。另外还可能需要进行电压的偏置，例如当信号源为双极性而ADC输入为单极性时（电压只能为正）。

如果正在转换的信号为周期信号，那么转换中的基本要求是，转换速度至少应为信号最高频率的2倍。这就是人们熟知的奈奎斯特采样准则（Nyquist sampling criterion）。如果不能满足该准则，那么将产生效果极差的信号降格，称为混叠（aliasing）（细节可参见参考文献1.1或信号处理方面的教材）。因此，需要使用反混叠滤波以确保满足奈奎斯特准则。

11.2.3 模拟多路选择器

如果存在多个输入，就需要使用模拟多路选择器（multiplexer）。另外也可以使用多个ADC，但这种方法不但昂贵而且占用空间。多路选择器是一种选择器开关，在任意时刻都可以从多个输入中选择某个输入与ADC相连。多路选择器由一系列半导体开关组合而成。重要的是，半导体开关并不是一种精密器件。尤其是在开关打开的时候，它具有内部串联电阻，大小从几十欧姆到几千欧姆不等。这会对数据采集过程产生影响，本书后面会介绍这一点。

308

11.2.4 采样与保持以及采集时间

由于多数ADC不能精确地转换变化中的电压，因此通常会使用*采样与保持*（sample and hold，S&H）电路。它对电压进行采样（类似于快照），并在转换持续时间内保持稳定。图11-3显示了一种简单实用的S&H电路，其核心只是一个半导体开关和电容器。当开关闭合时，电容器充电至输入电压V_S。此时，由于缓冲放大器增益为1，因此理想状态为$V_O=V_C=V_S$。当开关断开时，电荷停留在电容器上，V_C（继而V_O）保持为固定值。在实际情况下，电容器会存在一些漏电流，因此输出电压会发生漂移。这种电路有时也称为*跟踪与保持*（track and hold）电路，这是因为当开关闭合时，输出电压将跟随（或跟踪）输入。

这种简单电路存在的一个问题是，在信号通路上不可避免地会存在串联电阻。这用电路中的电阻器来表示。因此，当开关闭合时，电容电压V_C就不能立刻取得信号电压，而是以指数规律逐渐上升至信号电压。详细情况如图11-4中所示。电压上升规律如下：

$$V_C=V_S[1-\exp(-t/RC)] \tag{11-1}$$

从数据采集的观点来看，关键在于确保开关闭合之后使电压上升到足够接近于它的最终值，然后将开关断开（信号就被“保持”），此时就可以开始转换。V_C（继而V_O）上

升到它的可接受值所需的时间被称为采集时间(acquisition time)。

假设 V_C 必须上升至最终值 V_S 的 90%。那么,带入等式(11-1)可得:

$$0.9V_S = V_S[1-\exp(-t/RC)]$$

$$\exp(-t/RC) = 1-0.9$$

$$-t = RC\ln(0.1)$$

$$t = 2.3RC$$

上述分析结果如图 11-4 所示。然而,这样的要求并不严格。要确保数据转换中的较高精度,上述过程中所引入的误差应不高于半个 LSB 的对应值。因此,在 8 位转换中,所采集的电压值 V_C 需要等于或大于 $(511/512)V_S$,即 $0.9980V_S$。对于 10 位转换,则需要等于或大于 $(2047/2048)V_S$,即 $0.9995V_S$。

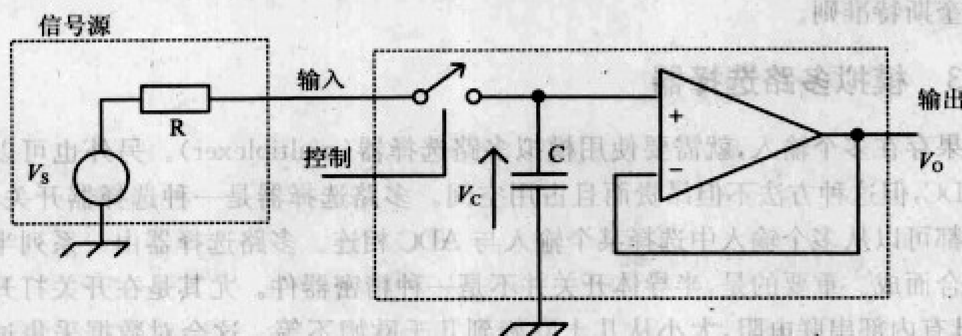


图 11-3　采样与保持电路的简单形式

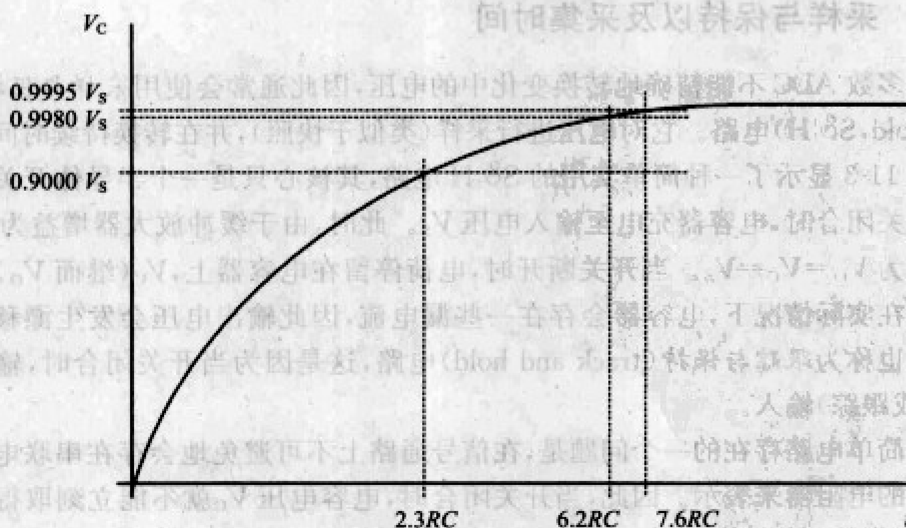


图 11-4　采集时间分析(未标明刻度)

将 10 位数值带入上述公式,同样的方法,可以得到:

$$-t = RC\ln(1/2048)$$

$$t = 7.6RC$$

最终求得的采集时间如图 11-4 所示。很明显，采集时间随电容、电阻的增大以及所需精度的提高而延长。本章后续部分将涉及这种计算的实际应用。

需要注意的是，多路选择器与 S&H 电路可以合并为一个整体，当多路选择器开关时，S&H 也随之开关，这是通常的做法。

310

11.2.5 时序及微处理器控制

通常使用微处理器或微控制器来控制数据采集系统。它们可以控制整个系统时序，包括对输入端口的选择、对所选信号的采样时间以及启动转换的时间。

一次转换过程可以用图 11-5 所示的流程图来表示。需要满足 2 个主要的时间要求——采集时间(S&H 的)和转换时间(ADC 的)。

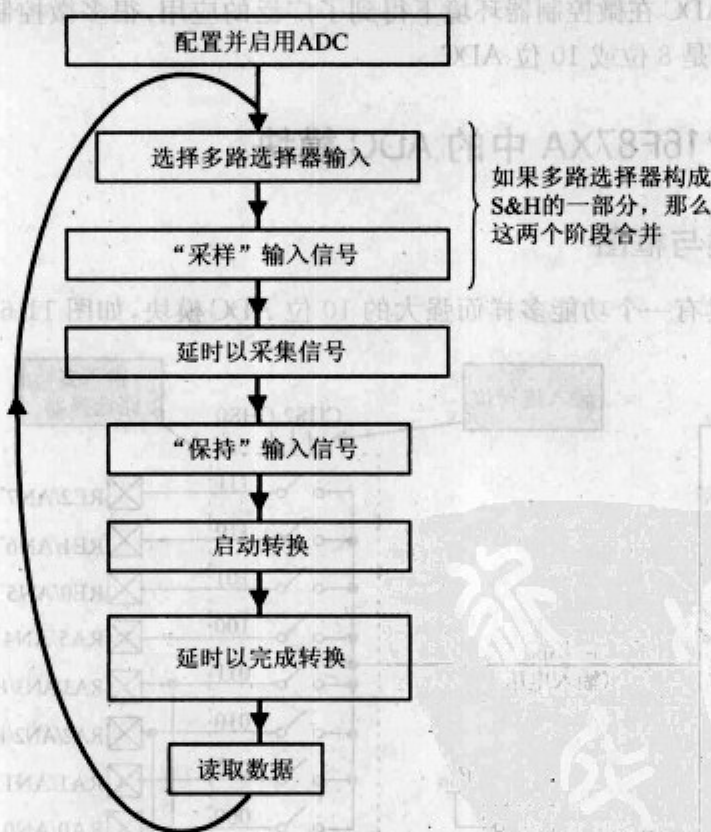


图 11-5 一次 A/D 转换的典型时序要求

一旦系统完成初始化，就可以对多路选择器的开关进行设置。之后 S&H 将启动采样过程。之后应延迟一段时间，时长应等于或大于 S&H 采集时间。之后，ADC 可以开始转换。同样地，转换过程需要一段有限的时间。ADC 将在转换完成后发出信号标志，然后微处理器就可以读取输出数据。

11.2.6 微控制器环境下的数据采集

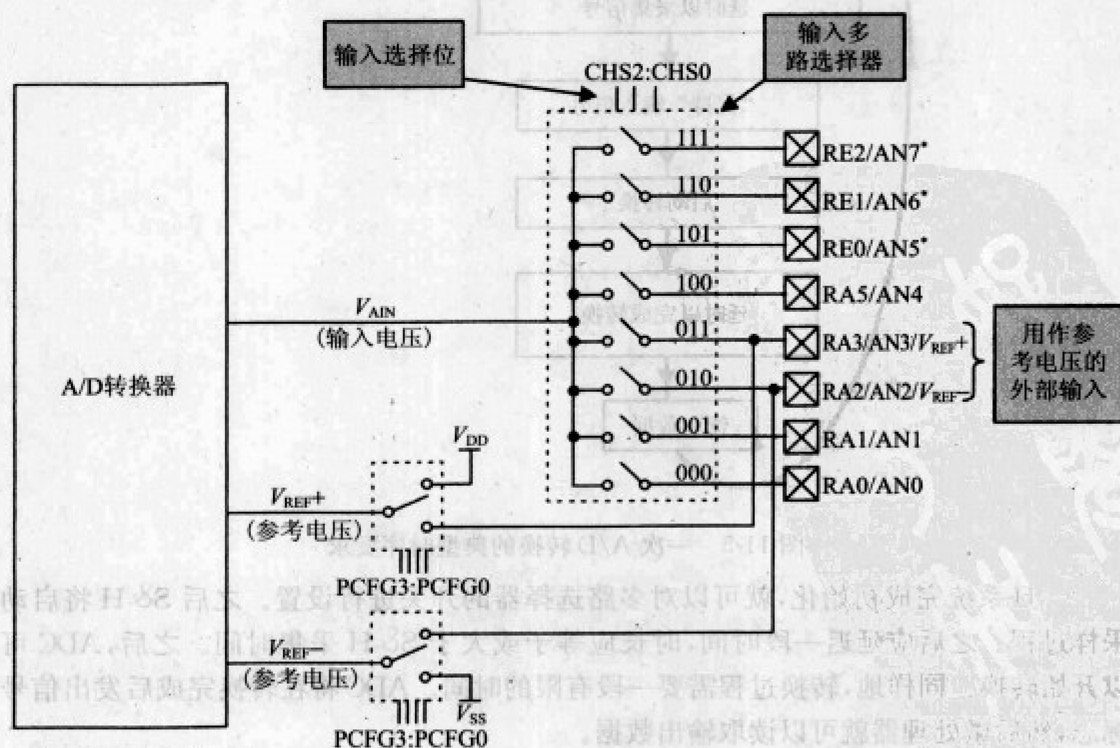
ADC是嵌入式系统不可缺少的模块，因此人们自然希望将ADC作为外围设备集成在微控制器中。然而我们必须认识到，由于ADC与微控制器在工作条件上的差别，311要实现二者之间的协同工作并不容易。ADC要达到较高的精度，必须工作于无噪声环境下（从电子学角度讲），需要提供优良而纯净的电源与地，并且应远离电磁干扰。而对于微控制器而言，作为一种数字器件，它会在每个电平切换沿上产生电压毛刺，从而降低电源与地的品质。它所造成的后果就是，当微控制器内部数字活动较为剧烈时，它会向周围散播局部干扰。因此，将ADC集成在微控制器内部至多只能是一种折中的方法，通常很难达到较高的精度。

尽管如此，ADC在微控制器环境下得到了广泛的应用，很多微控制器都提供片上ADC。它们通常是8位或10位ADC。

11.3 PIC® 16F87XA 中的ADC模块

11.3.1 概述与框图

16F87XA具有一个功能多样而强大的10位ADC模块，如图11-6所示。图中描



312

图 11-6 16F87XA 模数转换器（阴影框中所附标签为作者所加）

绘的内容是图 11-1 所示的整个数据采集系统的子集，包括 ADC、多路选择器，并且可以将电源电压作为参考电压来使用。ADC 中采用了特殊设计，以一种有趣的方式将采样与保持功能集成在其中，相关细节将在 11.3.3 节中进一步讨论。

位于图中靠右位置的输入多路选择器，具有 5 个通道（对于 16F873A 和 F875A）或 8 个通道（对于 16F874A 和 F876A）。这些输入通道与端口 A 的 6 个端口位中的 5 个以及端口 E 的 8 个端口位中的 3 个（对于 16F874 和 F876）共享引脚。端口 A 中仅位 4 未使用，它已经复用为重要的 Timer 0 输入。通过设置 SFR，可以将端口位灵活地配置为模拟输入或数字输入。

对于需要较高精度的应用，可以使用外部参考电压，器件为此提供有正连接与负连接引脚。器件所提供的负连接表明，参考电压并不一定要以系统地作为基准。对于较低成本、较低精度的转换，可以直接将电源电压用作参考电压。输入范围等于所选参考电压。

11.3.2 控制 ADC

ADC 由 2 个 SFR 进行控制：ADCON0（如图 11-7 所示）和 ADCON1（如图 11-8 所示）。转换结果放在另外 2 个 SFR 中：ADRESH 和 ADRESL。这 4 个寄存器都可以在图 7-6 中找到。另外，还有一些 SFR 会对 ADC 产生重要影响。其中包括 TRISA 和（对有 40 个引脚的器件而言）TRISE。任何用作模拟输入的端口位必须在这些寄存器中设置为输入。此外还要用到寄存器 PIR1 和 PIE1，分别包含有 ADC 中断标志位和中断使能位。

下面将按照可能的使用顺序对各个控制动作进行描述。

1. 开启

通过 ADCON0 中的 ADON 位可以开启或关闭 ADC。在不需要时关闭 ADC 可以略微降低能耗。

2. 设置转换速度

16F87XA ADC 的运行受控于 ADC 时钟，周期为 T_{AD}。完整的 10 位转换大约需要 12 个 T_{AD}周期，这与所选时钟源也有少许关系。用户可以有多个时钟频率可供选择。尽管在通常都希望转换尽快进行，但时钟频率有上限要求。对 16F87XA 而言，完成正确运行所需的最小时钟周期为 1.6μs（引自参考文献 7.1 中的 *Electrical Characteristics*），即频率为 625kHz。也就是说，最快的转换时间为 19.2μs。对于另一个极端，如果转换速度过慢，存储电容上将发生电荷泄漏，从而导致转换精度降低。因此，最好的方法是设置 ADC 时钟频率，使其周期等于或略小于 1.6μs。

通过 ADCON1 中的 ADCS2 以及 ADCON0 中的 ADCS1 和 ADCS0 可以控制对 ADC 时钟频率的选择，如图 11-7 所示。从图中可以看出，可以选择主时钟频率的不同分频比。另外，还可以选择专用的 RC 振荡器，其周期典型值为 4μs，范围是 2μs～6μs。

如果系统时钟较快，通常可以从系统时钟获取时钟源。但是，如果系统时钟较慢，

最好使用 RC 振荡器。这里，时钟振荡器"较快"与"较慢"的分界线大约是 500kHz。如果内部振荡器以此速度运行，ADC 所能获得的最快时钟频率为 250kHz。此时对应的周期为 4μs，等于 RC 振荡器的典型周期。如果主振荡器比上述频率慢，那么通常建议使用 RC 振荡器。

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	U-0	R/W-0
ADCS1	ADCS0	CHS2	CHS1	CHS0	GO/$\overline{\text{DONE}}$	—	ADON
位 7							位 0

位 7和位6 **ADCS1:ADCS0**：A/D转换时钟选择位（ADCON0位以黑体表示）

ADCON1 <ADCS2>	ADCON0 <ADCS1:ADCS0>	时钟转换
0	00	$F_{OSC}/2$
0	01	$F_{OSC}/8$
0	10	$F_{OSC}/32$
0	11	F_{RC}（时钟取自内部A/D RC振荡器）
1	00	$F_{OSC}/4$
1	01	$F_{OSC}/16$
1	10	$F_{OSC}/64$
1	11	F_{RC}（时钟取自内部A/D RC振荡器）

位 5~3 **CHS2:CHS0**：模拟通道选择位

000＝通道0（AN0）
001＝通道1（AN1）
010＝通道2（AN2）
011＝通道3（AN3）
100＝通道4（AN4）
101＝通道5（AN5）
110＝通道6（AN6）
111＝通道7（AN7）

注：PIC16F873A/876A器件仅实现了A/D通道0至通道4；未实现的选项被保留。在使用此类器件时不要选择任何未实现通道

位 2 **GO/$\overline{\text{DONE}}$**：A/D转换状态位

当ADON=1时

1=正在进行A/D转换（将该位置1则启动A/D转换，当A/D转换结束后由硬件自动清零）

0=未进行A/D转换

位 1 **未实现**：读作"0"

位 0 **ADON**：A/D开启位

1＝A/D转换器模块上电

0＝A/D转换器模块关闭，不消耗工作电流

图 11-7 ADCON0 寄存器（地址 $1F_H$）

3. 配置输入通道并选择参考电压

通过设置 ADCON1 中的 PGFC3 和 PGFC0 位可以定义输入端口位的使用方式。有必要仔细观察图 11-8。在输入通道和参考电压的选择上，图中所提供的众多选项确实令人印象深刻。既可以仅利用端口 A 的 1 个通道作为模拟输入（PGFC3：PGFC0＝1110），也可以同时使用所有 8 个模拟输入（PGFC3：PGFC0＝0000）。另外，在很多组

合中还可以使用外部参考电压。同样需要注意的是:对于任何用作模拟输入的端口引脚,必须在 TRIS 寄存器中将其设置为输入。否则,引脚将作为输出端口进行动作,所转换的信号是(未预料的)数字输出值。

R/W-0	R/W-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0
ADFM	ADCS2	—	—	PCFG3	PCFG2	PCFG1	PCFG0
位 7							位 0

位 7 **ADFM:A/D结果格式选择位**

1=右对齐。ADRESH寄存器的高6位读作“0”

0=左对齐,ADRESL寄存器的低6位读作“0”

位 6 **ADCS2:A/D转换时钟选择位(ADCON1位以阴影和黑体显示)**

ADCON1 <ADCS2>	ADCON0 <ADCS1:ADCS0>	时钟转换
0	00	$F_{OSC}/2$
0	01	$F_{OSC}/8$
0	10	$F_{OSC}/32$
0	11	F_{RC}(时钟取自内部A/D RC振荡器)
1	00	$F_{OSC}/4$
1	01	$F_{OSC}/16$
1	10	$F_{OSC}/64$
1	11	F_{RC}(时钟取自内部A/D RC振荡器)

位 5和位4 **未实现:读作“0”**

位 3和位0 **PCFG3:PCFG0:A/D端口配置控制位**

PCFG <3:0>	AN7	AN6	AN5	AN4	AN3	AN2	AN1	AN0	V_{REF+}	V_{REF-}	C/R
0000	A	A	A	A	A	A	A	A	V_{DD}	V_{SS}	8/0
0001	A	A	A	A	V_{REF+}	A	A	A	AN3	V_{SS}	7/1
0010	D	D	D	A	A	A	A	A	V_{DD}	V_{SS}	5/0
0011	D	D	D	A	V_{REF+}	A	A	A	AN3	V_{SS}	4/1
0100	D	D	D	D	A	D	A	A	V_{DD}	V_{SS}	3/0
0101	D	D	D	D	V_{REF+}	D	A	A	AN3	V_{SS}	2/1
011x	D	D	D	D	D	D	D	D	—	—	0/0
1000	A	A	A	A	V_{REF+}	V_{REF-}	A	A	AN3	AN2	6/2
1001	D	D	A	A	A	A	A	A	V_{DD}	V_{SS}	6/0
1010	D	D	A	A	V_{REF+}	A	A	A	AN3	V_{SS}	5/1
1011	D	D	A	A	V_{REF+}	V_{REF-}	A	A	AN3	AN2	4/2
1100	D	D	D	A	V_{REF+}	V_{REF-}	A	A	AN3	AN2	3/2
1101	D	D	D	D	V_{REF+}	V_{REF-}	A	A	AN3	AN2	2/2
1110	D	D	D	D	D	D	D	A	V_{DD}	V_{SS}	1/0
1111	D	D	D	D	V_{REF+}	V_{REF-}	D	A	AN3	AN2	1/2

A=模拟输入 D=数字I/O

C/R=模拟输入通道个数/ A/D参考电压个数

图 11-8 **ADCON1 寄存器(地址 $9F_H$)**

4. 选择输入通道

通过 ADCON0 中的通道选择位 CHS2 至 CHS0 可以选择输入通道。这些位用于确定图 11-6 中的哪个开关闭合。下面将会发现，选择输入通道通常是数据采集过程中的第 1 步。

5. 启动转换并在结束时给出标志

通过置位寄存器 ADCON0 中的 GO/$\overline{\text{GONE}}$位可以启动转换。转换结束时，该位由硬件恢复到 0，同时通过 ADC 中断标志 ADIF 给出转换结束信号，如图 7-10 所示。因此，要检测转换是否结束，可以检测 GO/$\overline{\text{GONE}}$位或 ADIF 位，也可以启用相应中断并在 ISR 中做出响应。

6. 对结果进行格式化

转换结果放置在寄存器 ADRESH 和 ADRESL 中。图 11-9 给出了 2 种可能的结果。结果可以按左对齐放置，高 8 位出现在 ADRESH 中。这适用于仅需要 8 位结果的情况，此时可以忽略 ADRESL 中的内容。在其他的多数情况下，使用右对齐方式更为有利。存放格式由 ADCON1 中的 ADFM 位来控制。

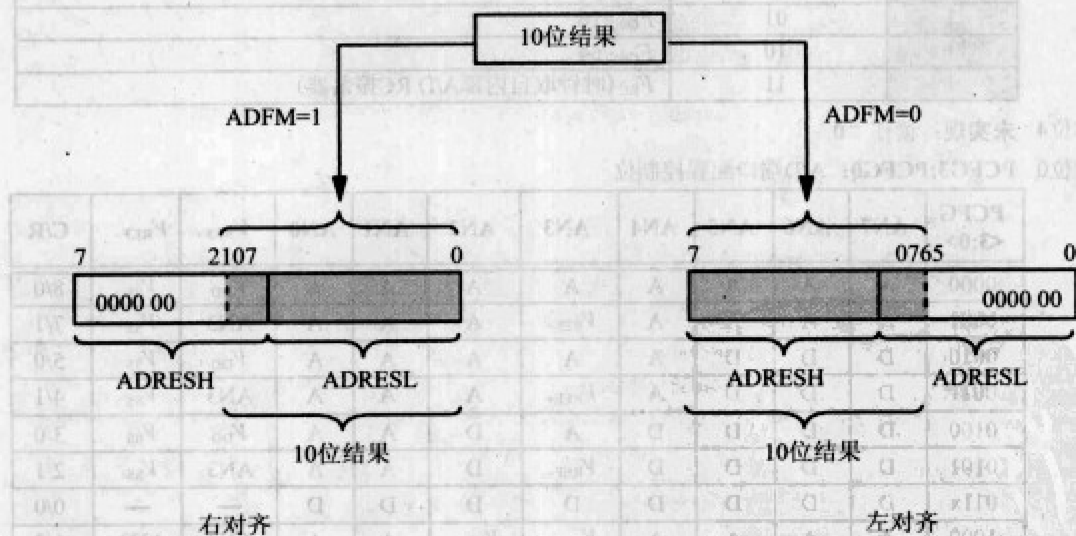


图 11-9 ADC 转换结果的存放格式

316

11.3.3 模拟输入模型

本章前面已经证明，对实际信号通路的理解对于理解和预测系统特性是很有必要的。图 11-10 就是这种 ADC 信号通路的框图——看起来是一门很深奥的课程！该图从信号传输的视角表示出图 11-6 中部分模块的真实情况。

信号源及其内部电阻显示在图的左边，相应模型为电压源串联内部电阻。信号源通过引脚 ANx 进入微控制器。存在微小输入电容(5pF)，且输入保护二极管以及其他输入电路明显具有一定的电压，从而向信号通路泄漏电流。之后，信号流过互连电阻

R_{IC}到达多路选择器开关。这是图 11-6 中的模拟输入多路选择器中的一个开关。图中还画出了开关的内部电阻 R_{SS}。采样开关的近似电阻值取决于电源电压，具体关系如图 11-10 右下部的小图中所示。可以看出，当电源电压为 5V 时，开关电阻近似为 7kΩ。

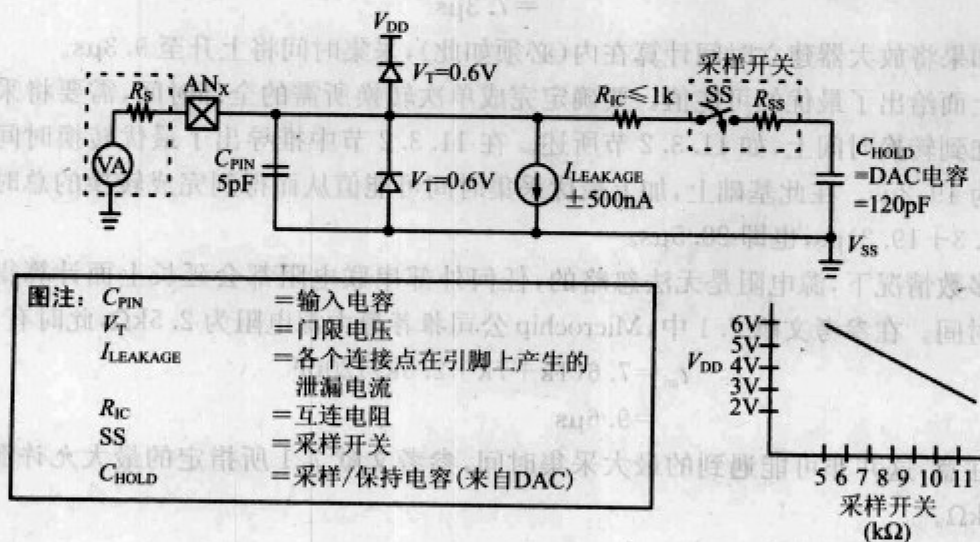


图 11-10 16F87XA ADC 输入模型

ADC 本身就是一种所谓的开关电容(见参考文献 1.1 以及第 5 章)。首先，这意味着 ADC 具有内部电容，在开始转换之前必须将其充电至输入电压。然而巧妙的是，这个电容还具有 S&H 电容的功能。对于不利的一面，必须首先将电容的所有 120pF 完全充电。

11.3.4 计算采集时间

在参考文献 7.1 中，Microchip 公司在采集时间 t_{ac} 的计算中定义了 3 个延时因素，即：

$$t_{ac}=\text{放大器的建立时间}+\text{保持电容的充电时间}+\text{温度系数} \tag{11-2}$$

参考文献中指定放大器的建立时间为 2μs。温度系数仅在温度高于 25℃的情况下使用，计算方法为：

$$\text{温度系数}=(\text{温度}-25℃)(0.05\mu s/℃)$$

可以看出，在 25℃以上每升高 10℃仅增加时延 0.5μs，因此多数情况下它所产生的影响是很小的。

影响采集时间的主要因素为电容充电时间，下面将对此进行分析。图 11-10 中的模拟输入模型可以和图 11-3 中的 S&H 图以及式(11-1)联系起来考虑。分析中，可以忽略输入漏电流以及较小的输入电容的影响。16F87XA ADC 所对应的图 11-3 中的 R、C 的实际值可以从图 11-10 中获得。R 为($R_{SS}+R_{IC}+R_S$)或($1k+7k+R_S$)，电源电压为 5V，C 为 120pF。与图 11-4 所对应的计算表明，要获得 10 位精度，所需的采集时

间为 7.6RC。将这些值代入公式，首先假设源电阻可以忽略，于是有：

$$t_{ac}=7.6RC$$
$$=7.6(1k+7k)120pF$$
$$=7.3\mu s$$

如果将放大器建立时间计算在内(必须如此)，采集时间将上升至 9.3μs。

上面给出了最优的可能值。要确定完成单次转换所需的全部时间，需要将采集时间添加到转换时间上，如 11.3.2 节所述。在 11.3.2 节中推导出了最优转换时间的可能值为 19.2μs。在此基础上，加上最优采集时间可能值从而得到完成转换的总时间为 (2+7.3+19.2)μs，也即 28.5μs。

多数情况下，源电阻是无法忽略的，任何外部串联电阻都会延长上面计算得到的采集时间。在参考文献 7.1 中，Microchip 公司推荐最大源电阻为 2.5kΩ，此时有：

$$t_{ac}=7.6(1k+7k+2.5k)120pF$$
$$=9.6\mu s$$

注意，这并非可能遇到的最大采集时间，参考文献 7.1 所指定的最大允许源电阻为 10kΩ。

318

11.3.5 重复转换

当一次转换完成之后，转换器将在启动新的转换周期之前等待 $2\times T_{AD}$的时间。一旦这段时间结束，就可以在同一通道上开始新的转换，或者选择新的通道进行转换(必须事先选择该通道)。

在 11.3.4 节中已经得到最优转换时间的可能值为 28.5μs。如果再加上 $2\times T_{AD}$的时间(最快时为 3.2μs)，那么完整的转换周期时间为 31.7μs。这意味着在连续转换中的最大采样速率为 30kHz。但是需要注意，上述计算中并没有考虑软件负荷，而软件负荷会降低转换速率。

11.3.6 综合权衡转换速率与转换精度

按照现在的标准，上述计算得到的转换时间并不是特别快。某些情况下，需要更短的转换时间。虽然可以使用外部 ADC 来解决这一问题，但也可以考虑是否使用全 10 位精度。如果不使用全 10 位精度，就可以缩短转换时间。参考文献 11.1 描述了这样一种技术：首先在有效的 ADC 时钟频率下启动转换，然后在转换过程中将其切换到更高的速率上去(该速率与时钟要求不一致)。在频率切换之前转换得到的高阶位有效并可以使用，而在频率切换之后转换得到的数位则无效。这样就可以以较快的转换速率得到较低的转换精度。但是，需要通过编程来控制频率切换的发生时间。

另一种方式就是缩短采集时间，如图 11-4 所示，这样采集过程也许只能得到 8 位精度。然后，就可以实现完整过程的转换。另外，也可以按照上述方法对 ADC 时钟频率进行切换。

11.4 在 Derbot 测光程序中应用 ADC

Derbot AGV 具有 3 个光敏电阻(light-dependent resistor,LDR),用作光传感器,如图 A3-1 所示。其中 2 个在小车前侧,另一个在小车后侧。Derbot 使用 16F873A ADC 来测量光强,在测光程序和寻光程序中均采用了这种方法。

例程 11-1 给出了程序 Dbt_light_meter 的部分片断,完整版本可以在本书附属资源中查找。程序读取 LDR 输出值并将其显示在手动控制器单元的 LCD 上。然而,在这个过程中需要进行一些数据处理。必须将 ADC 的转换结果比例缩放到真实的电压读数,然后转化为 LCD 可以显示的格式。因此,需要先将电压读数转换为 BCD(Binary Coded Decimal,二进制编码的十进制)码,然后转换为 ASCII 码。最终得到的字符则被传送到 LCD。下面将详细分析这一过程。

例程 11-1 在 Derbot 测光程序中应用 ADC

```
;****************************************************************************
;Dbt_light_meter
;Dbt reads LDR values to 10-bit resolution, scales to a voltage reading,
;converts to 4 digits of BCD, and displays on Hand Controller LCD.
;Rear LDR is scaled to be a true millivolt reading, displayed to tens of mV.
;Requires Hand Controller loaded with corresponding program.
;TJW 19.7.05                                               Tested 20.7.05
;****************************************************************************
...
(early program sections omitted)
...
        bsf     status,rp0
        movlw   B'00001011'  ;set port A bits according to their function,
        movwf   trisa            ;ADC channels set as inputs
        movlw   B'10000100'  ;select port A bits 0,1,3 for analog input
        movwf   adcon1           ;right justify result
...
        bcf     status,rp0
        movlw   B'01000001'   ;set up ADC: clock Fosc/8, switch ADC on but not
                                                          ;converting,
        movwf   adcon0        ;input channel selection currently irrelevant
...
;read and store ldrs
main_loop movlw B'01000001' ;select channel 0 as input (left front ldr)
        movwf   adcon0
        call    delay20u      ;acquisition time
        bsf     adcon0,go     ;start conversion
        btfsc   adcon0,go_done ;wait for conversion to complete
        goto    $-1
        movf    adresh,0      ;read and store ADC output data, high byte
        movwf   ldr_left_hi
        bsf     status,rp0
        movf    adresl,0      ;read and store ADC output data, low byte
        bcf     status,rp0
        movwf   ldr_left_lo
```

```
;select channel 1 (right ldr)
      movlw  B'01001001'
      movwf  adcon0        ;select channel 1 as input (right front ldr)
      call   delay20u
...
(samples other two LDRs)
...
(scales, converts to BCD, and outputs values to diplay)
...
      goto   main_loop
...
```

11.4.1 ADC 的配置

例程给出了 ADC 控制寄存器 ADCON0 和 ADCON1 的初始设置。从图 A3-1 可以看出，右边的 LDR 与端口 A 的位 0 相连，左边的 LDR 与端口 A 的位 1 相连，而后边的 LDR 与端口 A 的位 3 相连。没有外部参考电压可供使用，因此将电源作为参考电压。在这种连接下，需要将 ADCON1(见图 11-8)的 PCFG 位设置为 0100。

由于主时钟频率为 250ns(4MHz)，而要求的最小 T_{AD} 为 1.6μs，因此至少需要将时钟频率分频 8 倍。此分频值所对应的 T_{AD} 为 2μs。转换结果选择右对齐方式。

11.4.2 采集时间

要了解数据采集的具体操作，可以查看例程的后续部分。

另外，需要计算最坏情况下的采集时间，最坏情况是指环境光较弱、LDR 电阻值较大的情况。可以查看 LDR 的相关数据[8.5]。最坏情况下，LDR 电阻值很大，源电阻与 LDR 串联，将达到近 10kΩ 的阻值。注意，这是源电阻的最大允许上限值。但是，在正常的光线条件下，源电阻将小得多。由图 11-4 可知，采集时间由下式给出：

$$\begin{aligned} t &= 7.6RC \\ &= 7.6(10\text{k}+1\text{k}+7\text{k})120\text{pF} \\ &= 16.4\mu\text{s} \end{aligned}$$

另外加上放大器的建立时间，得到 18.4μs。

11.4.3 数据转换

例程中 main_loop 标号以下的内容是 ADC 的实际操作过程。程序对 3 个 LDR 逐个采样。对于第 1 个 LDR(左前侧)，程序首先选择相应的输入通道，并将 ADCON0 中的 GO/$\overline{\text{GONE}}$置为 0。在信号采集之前引入 20μs 延时，稍大于上面求得的最坏情况下的延时长度 18.4μs。然后，通过将 GO/$\overline{\text{GONE}}$位置 1 启动实际转换。通过测试该位等待转换完成。转换结果从 ADRESH 和 ADRESL 传送至两个存储位置，程序继续采样下一个输入。例程中没有给出后续的数据处理过程，但本章稍后将对此加以描述。

11.5 一些简单的数据处理技术

在多数情况下，程序一旦获得数据，就需要以某种方式加以处理。这包括简单的加减，就像前面所做的那样，另外还需要其他的算术运算，如乘除等。由于数据处理要求的提高，汇编程序的复杂性也必然随之上升，从而产生了很强的驱动力促使人们转向高级编程语言。然而，并没有必要用汇编语言编写标准的数学运算程序或子例程。有很多现成程序可用，可以查看参考文献 11.2。

319~321

11.5.1 定点与浮点算术

前面已经反复用到这样的事实：n 位二进制数可以表示 $0\sim2^n-1$ 范围内的任何整数。例如，8 位数的表示范围为 $0\sim255_D$，12 位数的表示范围为 $0\sim4095_D$，16 位数的表示范围为 $0\sim65535_D$。要表达更大的数，只需增加位数即可。另外，通过插入二进制小数点，甚至可以通过这种方式来表示小数。此时，二进制小数点右侧的数位代表 2 的负数次幂。从图 11-11 的示例中可以看出，二进制数 1101.11 对应于十进制数 13.75_D。如果在所有用到的数字中，二进制小数点始终固定在同一位置（事实上只是想象在这个位置上有一个二进制小数点），那么就可以在这些数字集合上执行一些算术运算。

这种二进制数字表示方法称为*定点*（fixed point）。可以不使用二进制小数点，也可以假设它放置在某个固定位置。这种表示方法可以满足整型数和非整型数的表示要求，但却不能表示较大范围内的数（即覆盖极小数到极大数的整个范围）。例如，图 11-11所示的 6 位定点数可以表示的最小非零数为 0.25_D，最大数为 15.75_D。它不能表示 0.0004_D或 2.3×10^6。

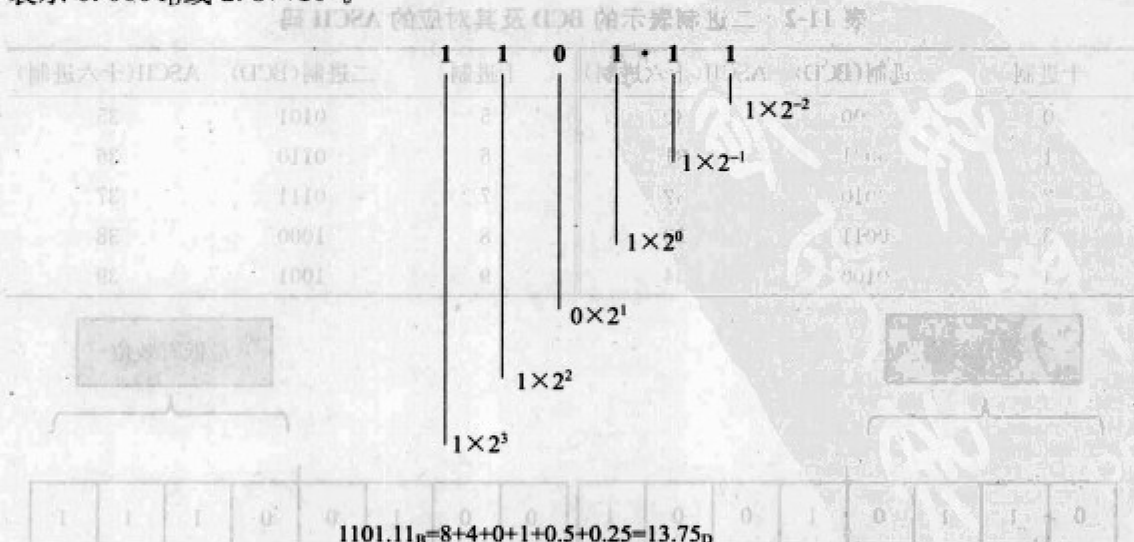


图 11-11 二进制小数

要解决上述问题，可以使用浮点(floating-point)表示法。它使用符号位、尾数和指数来表示某个数字。它可以表示很大范围内的数字，但处理起来比较复杂。虽然汇编语言也提供有浮点程序，但它在高级语言中则更为普遍，用于所有的精确计算。本章将只使用定点算术。

322

11.5.2 二进制数向 BCD 码的转换

虽然所有的定点算术都是以二进制形式完成的，但是只要涉及人机交互，人们都更倾向于使用十进制数。那么，应该如何在二进制域与十进制域之间进行数据转换呢?

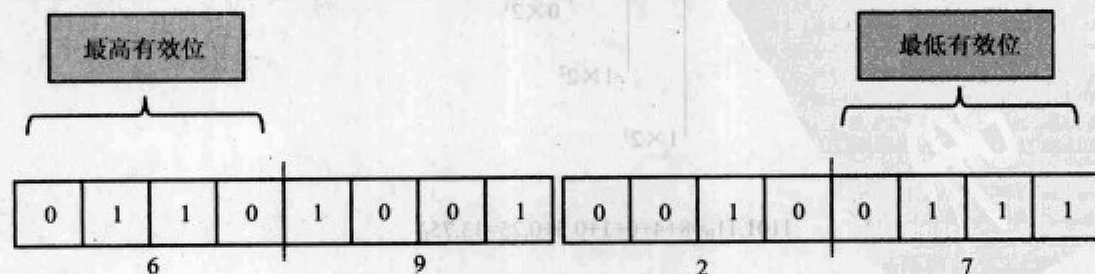
二进制与十进制之间的中转站就是 BCD(Binary Coded Decimal，二进制编码的十进制)。如表 11-2 所示，它用 4 位二进制数来表示 1 位十进制数。BCD 与十六进制之间的区别仅仅在于，在 BCD 中不能使用 A_H、B_H、C_H、D_H、E_H、F_H的等价二进制数。因此，表 11-2 给出了所有合法的 BCD 码，分别表示 1 位十进制数。表中还给出了每个数字所对应的 ASCII 码(American Standard Code for Information Interchange，美国信息交换标准码)。对于数字字符而言，可将其直接看作单字节码，其中数字本身占据低位，而高位为 3。

使用这种简单的编码方式，可以用 BCD 码来表示多位十进制数。通常使用压缩 BCD 码，1 个字节表示 2 位十进制数。图 11-12 给出了相关示例。

虽然可以采用这种简单的表示方式，但仍然无法在 BCD 和二进制之间实现完全的直接转换。但可以使用一些标准算法，例如参考文献 11.3 所用到的。Derbot 测光程序所用到的子例程就取自该参考文献，用于将 16 位二进制数转换为 5 位 BCD 码，用于为显示设备准备数据。具体算法可以在很多讲述计算机算法的书籍中找到，也可以查看参考文献 1.1。

表 11-2 二进制表示的 BCD 及其对应的 ASCII 码

十进制	二进制(BCD)	ASCII(十六进制)	十进制	二进制(BCD)	ASCII(十六进制)
0	0000	30	5	0101	35
1	0001	31	6	0110	36
2	0010	32	7	0111	37
3	0011	33	8	1000	38
4	0100	34	9	1001	39



323

图 11-12 十进制数 6927 的压缩 BCD 表示

11.5.3 乘法

在使用较多的计算机算术中，乘法是仅次于加减运算的最为常见的算术运算。某些处理器包含有硬件乘法器以及相应的乘法指令。而对于较为简单的器件，如 PIC 16 系列，乘法运算必须通过软件程序来实现。完成乘法运算的标准算法是反复的移位加(shift and add)的过程[1.1]。另外还有大量标准程序可供使用，包括定点运算与浮点运算。定点运算程序的大小取决于所用的字长。具有较长字长的乘法程序自然需要耗费较长的运行时间。参考文献 11.2 提供了定点乘法程序，适用于 8 位×8 位(16 位结果)、8 位×16 位(24 位结果)、16 位×16 位直到 32 位×32 位(64 位结果)的乘法运算。

Derbot 测光程序中用到的 16 位×16 位乘法子例程取自参考文献 11.3，下面将加以介绍。

11.5.4 比例缩放与 Derbot 测光示例

乘法运算通常用于对输入数据进行比例缩放，将其转换为标准单位。Derbot 对此给出了有趣的示例。

Derbot 使用 5V 电源电压作为基准，因此 10 位精度就对应(5/1024)＝4.883mV。当使用这种参考电压时，ADC 的最低有效位就等价于 4.883mV。如果将 ADC 输出值乘 4.883，就得到了真实的毫伏读数。但是引入并跟踪一个二进制小数点并不方便。可以采用这种方式：首先将该小数放大一个二进制幂的倍数，然后，在完成乘法运算之后再除以同一个数。上述过程是很容易实现的，二进制除法只是简单地将数字右移若干位，并去掉较低的有效位。

例如，在 Derbot 测光程序中需要计算：

$$\text{ADC 输出值} \times 4.883 = \text{毫伏读数}$$

虽然无法乘 4.883，但注意到 4.883×256＝1250.048，因此可以用整数来计算乘法：

$$\text{ADC 输出值} \times 1250 = \text{中间结果}$$

ADC 输出值为 10 位数，而 1250_D(即 $04E2_H$)为 11 位数，因此计算结果最多为 21 位。这就决定了所需乘法程序的类型。

下面需要将中间结果除以 256，可以直接去掉低位字节即可，这样就完成了比例缩放。最终得到的就是以毫伏为单位的 ADC 输入电压。

也许有人会问，上面的乘数 256 是如何选出来的？为什么不是 128 或 512 呢？这取决于所引入的舍入误差。将 1250.048 简单取作 1250 所带来的舍入误差是很小的，远低于 0.05%，因此在处理 10 位数字时是可以接受的。事实上，用 128 作乘数也是可以的，但在最后做除法时，就不能方便地通过去掉低位字节来实现了。

注意，这些仅仅只是一种变通的方法。同样地，也可以将 4.883 直接转换为二进制数，得到 100.11100010，即 $4.E2_H$。将它和上面的乘数进行比较，就可以很快地发

324

现，上述 2 种方法在本质上是等价的。

例程 11-2 用到了上述计算过程，它对后侧的 LDR 值进行缩放。其中用到 2 个子例程：**mult16x16** 用于将 2 个 16 位数相乘，**Bin2BCD16** 用于将 16 位数转换为 5 位 BCD 输出值。例程给出了 ADC 转换完成之后的程序段。原先保存在存储单元 **ldr_rear_hi** 和 **ldr_rear_lo** 中的 ADC 转换结果被传送至 **aargb0** 和 **aargb1**，用作乘法子例程的一个 16 位输入。由 **bargb0** 和 **bargb1** 组成的字用作子例程的另一个输入。其中装载了单字 $04E2_H$，即与 1250_D 等价的十六进制数。然后调用 **mult16x16** 子例程，结果放置在 **aargb0：aargb1：aargb2：aargb3** 中。由于结果最多为 21 位，因此最高字节 **aargb0** 将为空并被忽略。类似地，最低字节也被忽略，等价于将结果除以 256。紧靠该字节上方的字节 **aargb2** 就称为所需结果的低位字节，因而被传送至子例程 **Bin2BCD16** 的 16 位输入的低字节（**templ**）。类似地，将 **aargb1** 传送至高字节 **temph**。然后调用 **Bin2BCD16** 子例程。该子例程执行完成之后，可以立刻调用子例程 **four_dig_disp** 将输出字节传送至 LCD。该子例程在 I^2C 连接上逐个发送 BCD 字符，用于手动控制器 LCD 的显示。

在本书附属资源的完整程序清单中可以查看所有子例程。

例程 11-2　Derbot 测光程序中对后侧 LDR 的数据处理顺序

```
(ADC conversions have been undertaken for all three LDRs)
...
;scale rear
      movf    ldr_rear_hi,0    ;get higher byte of ADC conversion result
      movwf   aargb0           ;this is multiplier higher byte
      movf    ldr_rear_lo,0
      movwf   aargb1           ;this is multiplier lower byte
      movlw   04               ;higher byte of scaling factor
      movwf   bargb0
      movlw   0e2              ;lower byte of scaling factor
      movwf   bargb1
      call    mult16x16        ;call the multiply subroutine
      movf    aargb1,0         ;discard highest and lowest bytes
      movwf   temph            ;second highest byte for BCD conversion
      movf    aargb2,0         ;third highest byte for BCD conversion
      movwf   templ
      call    Bin2BCD16        ;call Binary to BCD conversion subroutine
      call    four_dig_disp    ;call subroutine which sends BCD bytes to display
...
```

325

11.5.5　使用参考电压实现比例缩放

在 Derbot 例程中，ADC 的输出值需要立刻进行比例缩放，从而获得真实电压读数，这显得有些繁琐，并且在程序执行过程中使用乘法程序必然是比较耗时的。

某些情况下，合理地选择参考电压，可以有效地简化后续计算。例如，如果 Derbot ADC 选择 4.096V 的参考电压，那么输出值的 1LSB 将精确代表 4mV，这样就不需要进

行复杂的比例缩放。类似地，如果参考电压为 1.024V，那么输出值的 1LSB 就精确代表 1mV。采用类似的参考电压可以满足特定目的的要求。

11.6 Derbot 寻光程序

这个程序给出了 16F873A ADC 的另一个应用实例。通过比较 3 个 LDR 的测量值，AGV 可以找到最强的光源并向该方向移动。它的移动速度取决于各个传感器之间的光强差，当 3 个传感器所感受到的光强近似相同时，AGV 将停止移动。由于只需要对测量值进行比较，因此使用时不需要很高的精确度。因而在程序中只使用了 8 位 ADC 结果。相应的结果处理方式也很方便，只需将结果左对齐（如图 11-9 所示），然后取出寄存器 ADRESH 中的 8 位数据。涉及的设置变更可以查看例程中对 ADCON1 的设置。

由于仅使用 8 位结果，这就为缩短采集时间提供了可能，理论上采集时间缩短为 6.2*RC*（见图 11-4）。由于程序只有当环境光线达到一定强度时才进行操作，因此估计 LDR 电阻为 10kΩ，将最大源电阻修改为 5kΩ。ADC 输入电容仍然是 120pF，ADC 内部串联电阻也维持不变。由此得到采集时间为 6.2×13k×120pF，即 9.7μs。例程使用了 11μs 的延时子例程。

例程 11-3 在 Derbot 寻光程序中应用 ADC

```
;**********************************************************************
;Dbt_light_seek
;Derbot seeks light. PWM applied. Speed is dependent on
;light difference (front to back), so Derbot comes to a
;halt when light difference is minimal. Microswitches used
;for bump detection.
;TJW 19.5.05                                    Tested 19.5.05
;**********************************************************************
...
(initial program sections omitted)
...
        bsf     status,rp0    ;select memory bank 1
        movlw   B'00001011'   ;set port A bits so that ADC bits are input
        movwf   trisa
        ...
        movlw   B'00000100'   ;Set up ADC, left justified result,
        movwf   adcon1            ;port A bits 0,1,3,for analog input
        ...
        bcf     status,rp0    ;select bank 0
        ...
        movlw   B'01000001'   ;Set up ADC, clock Fosc/8, ADC on but not running
        movwf   adcon0            ;input channel selection currently irrelevant
        ...
;initiate a conversion
        movlw   B'01001001'   ;select channel 1 (right ldr)
        movwf   adcon0
        call    delay11u
```

326

```
        bsf     adcon0,go      ;start conversion
adc2    btfsc   adcon0,go      ;wait for conversion to complete
        goto    adc2
        movf    adresh,0
        movwf   ldr_rt
...
```

图 11-13 给出了实际的控制算法，大约每 200ms 重复一次。完整清单可以在本书附属资源中找到。程序首先读取每个 LDR 值，然后进行一些简单的取平均和取差值的计算，用于前向速度的计算。然后确定最亮的 LDR 并执行相应动作。如果左前方或右前方最亮，它就向前移动并转向较亮的方向，移动和转向速度取决于相关光强。但是，如果后侧最亮，它将以固定速度在原地旋转，旋转方向朝向前方较亮的一侧。如果在它所进入的地点中，所有 LDR 感受到的光强相似，那么它将减速。如果所有 LDR 都处在近似等级的光强下，AGV 将停止移动。

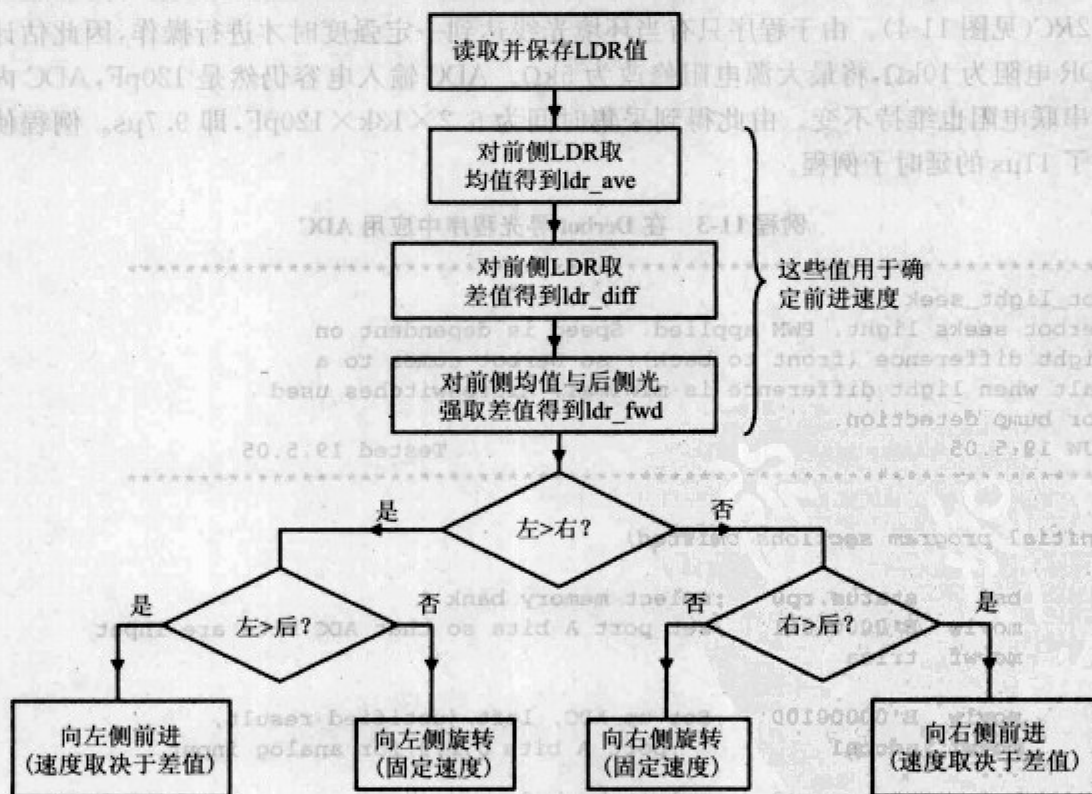


图 11-13 Derbot 寻光程序的框图

11.7 比较器模块

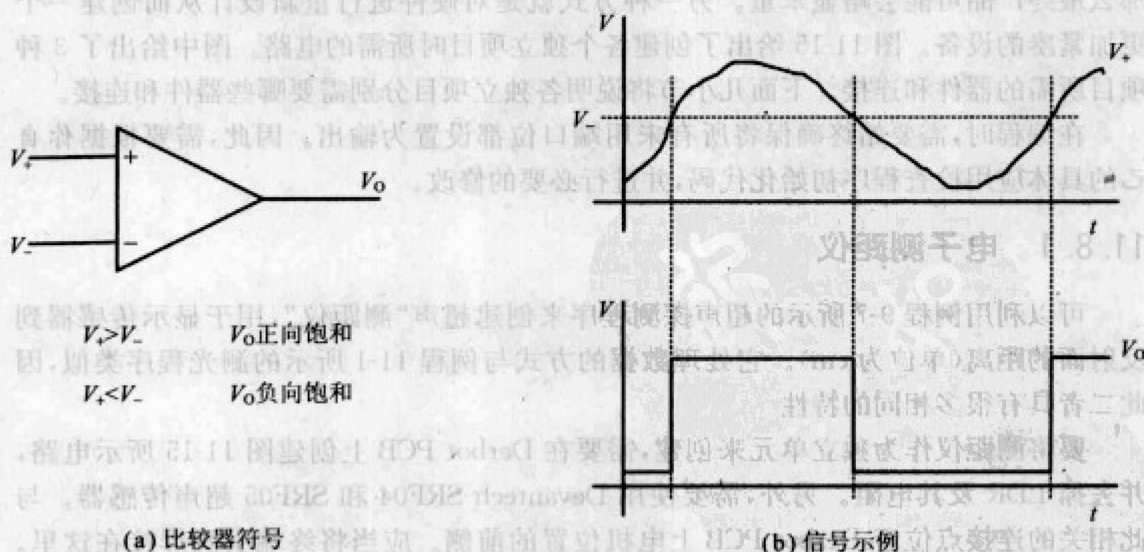
上面已经对 ADC 的复杂特性进行了分析，在本章的最后，将介绍一种联系模拟世界与数字世界的最简单接口——比较器。这是一种重要的电路元件，其行为有点类似 1 位 ADC，通常用于比较输入电压与参考电压的大小。如果输入电压高于参考电压，比较器输出高电平，反之则输出低电平。在嵌入式环境下，有很多应用需要使用比较器。具体包括：对降格的数字信号进行恢复（这种情况下，它们的用途非常类似于施密特触发器）、测试电池电压、在温度或其他变量达到特定值时进行报警，等等。

11.7.1 比较器动作概述

比较器如图 11-14a 中所示。比较器简单地对 2 个输入（图中为 V_+ 和 V_-）进行比较。如果 V_+ 大于 V_-，那么比较器就输出正电压，通常为“饱和”电压值。如果 V_+ 小于 V_-，那么比较器就输出负电压或零电压。通过合理设计输出电路，可以方便地将输出电压值转换为可识别的逻辑值。

327 ~ 328

图 11-14b 给出了输入电压与输出电压的示例。V_- 被设置为固定电压，以虚线表示，而 V_+ 则是可变的。一旦 V_+ 高于 V_-，输出将达到逻辑 1，反之则达到逻辑 0。



(a) 比较器符号　　(b) 信号示例

图 11-14　比较器

11.7.2 16F87XA 的比较器与参考电压

16F87XA 有 2 个比较器，与 ADC 输入端共用输入引脚。比较器 1 输入端为 AN0 和 AN3，比较器 2 的输入端为 AN1 和 AN2。它们由寄存器 CMCON 控制，在图 7-6 中位于地址 $9C_H$ 处。它们可以有各种不同的配置。例如，比较器输出可以向外引到 RA4

和 RA5 引脚上，也可以直接存放在 CMCON 寄存器的相关位中。这些特性可以在 Microchip 数据手册[7.1]中方便地查到，这里就不列举了。重要的是，比较器还可以形成中断源。任一比较器状态的改变都将置位 CMIF 标志位，如图 7-10 所示。

16F873A 还具有一个参考电压模块，由 CVRCON 寄存器（地址 $9D_H$）进行控制。它是一个电阻阵列，一端与电源电压相连（通过开关晶体管），这样就可以选择不同的电压值作为参考电压。参考电压可以用作 1 个或 2 个比较器的输入，并且可以通过 16F873A 的引脚 4 输出。此时，可以将其用作非常简单的数/模转换输出。

11.8 将 Derbot 电路用于其他测量

现在，我们已经了解了如何采集输入模拟电压、如何对采集到的数据进行处理然后进行显示。这是一个很重要的学习阶段，其中涉及的内容是很多测量系统的基础。本节将介绍如何利用 Derbot PCB 和手动控制器创建某些测量工具，并将用到前面提到的一些程序。手动控制器与 Derbot“总线”连接器相连，连接器的实际位置在微控制器 IC 固定插座的下方。需要事先为手动控制器装载标准程序，也即例程 10-3 所对应的程序清单。

这些测量项目可以创建为独立设备，也可以集成到 AGV 中。如果独立创建项目，那么最终产品可能会略显笨重。另一种方式就是对硬件进行重新设计从而创建一个更加紧凑的设备。图 11-15 给出了创建各个独立项目时所需的电路。图中给出了 3 种项目所需的器件和连接。下面几小节将说明各独立项目分别需要哪些器件和连接。

在编程时，需要始终确保将所有未用端口位都设置为输出。因此，需要根据你自己的具体应用检查程序初始化代码，并进行必要的修改。

11.8.1 电子测距仪

可以利用例程 9-7 所示的超声探测程序来创建超声“测距仪”，用于显示传感器到反射面的距离（单位为 cm）。它处理数据的方式与例程 11-1 所示的测光程序类似，因此二者具有很多相同的特性。

要将测距仪作为独立单元来创建，需要在 Derbot PCB 上创建图 11-15 所示电路，并去掉 LDR 及其电阻。另外，需要使用 Devantech SRF04 和 SRF05 超声传感器。与此相关的连接点位于 Derbot PCB 上电机位置的前侧。应当将终端引脚焊接在这里。应用传感器数据[8.7]，将 0V、5V、脉冲和回波连接到对应引脚上，然后用胶水将传感器粘贴在你所选定的 PCB 位置上。

完成上述创建过程，并围绕 **Dbt_US_Test** 程序建立 MPLAB® 项目，部分程序显示在例程 9-7 中。针对具体应用对初始化代码进行修改，将所有未用端口位设置为输出。编译连接程序然后下载至微控制器。将传感器对准某个反射面，传感器与该表面之间的距离将连续不断地显示在手动控制器显示屏上（单位为 cm）。

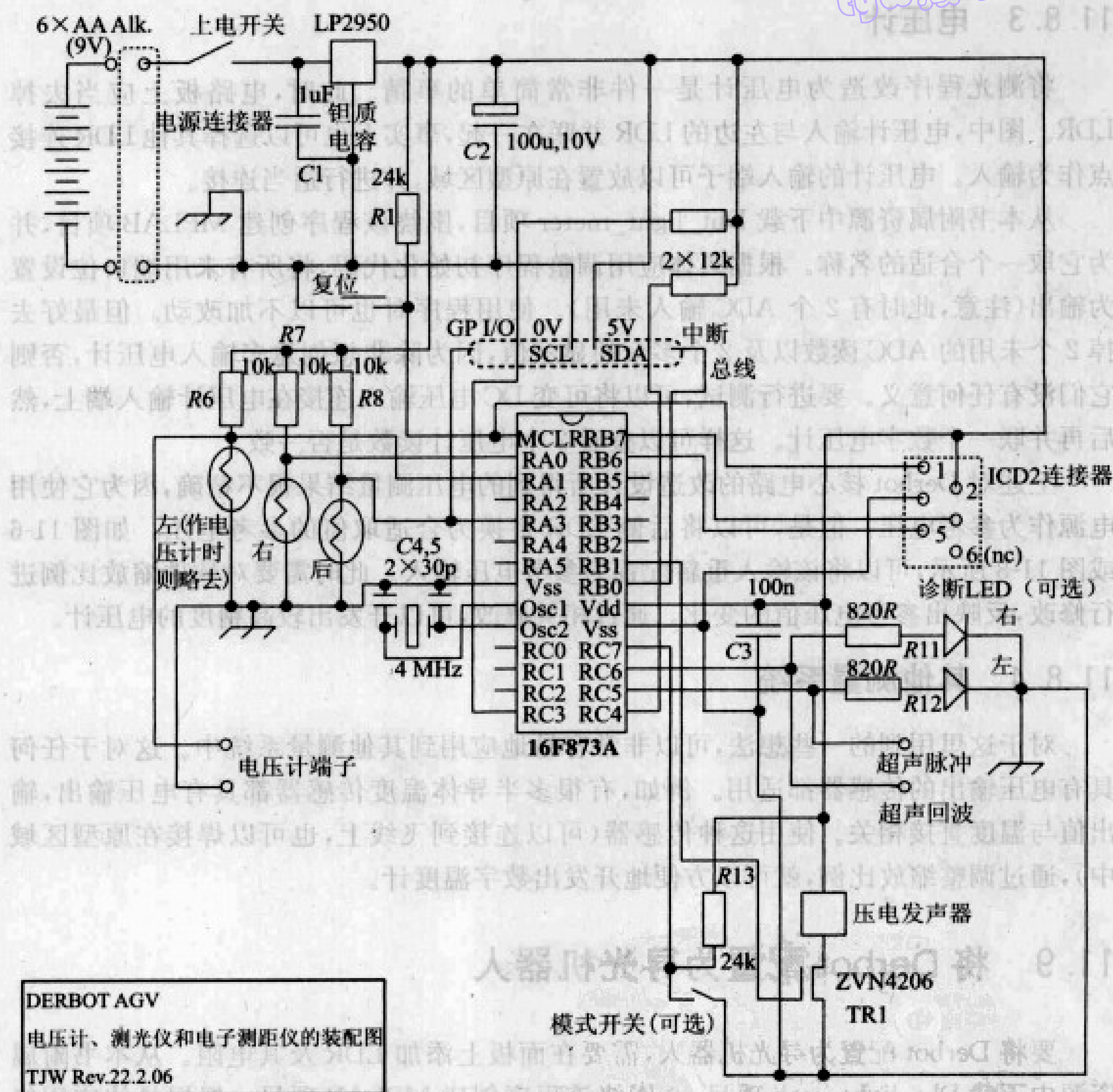


图 11-15 用作电压计、测光仪和电子测距仪的 Derbot 装配图

330

11.8.2 测光仪

要创建测光仪，需要去掉图 11-15 中的超声传感器。做好电路之后，从本书附属资源中下载 **Dbt_light_meter** 项目，并围绕该程序创建 MPLAB 项目。根据具体应用调整程序初始化代码，将所有未用端口位设置为输出。编译连接程序然后下载至微控制器。运行程序，此时可以看到显示屏上将给出所有 LDR 读数。前面已经提到，每个读数都以毫伏为单位，数值大小反映光的强弱。注意，当光强增加时，读数将减小。

11.8.3 电压计

将测光程序改造为电压计是一件非常简单的事情。此时，电路板上应当去掉LDR。图中，电压计输入与左边的LDR并联在一起，事实上也可以选择其他LDR连接点作为输入。电压计的输入端子可以放置在原型区域，并进行适当连接。

从本书附属资源中下载Dbt_light_meter项目，围绕该程序创建MPLAB项目，并为它取一个合适的名称。根据具体应用调整程序初始化代码，将所有未用端口位设置为输出(注意，此时有2个ADC输入未用)。使用程序时也可以不加改动。但最好去掉2个未用的ADC读数以及2个多余的显示值，因为除非想创建多输入电压计，否则它们没有任何意义。要进行测试，可以将可变DC电压输入连接在电压计输入端上，然后再并联一个数字电压计。这样可以查看2个电压计读数是否一致。

上述对Derbot核心电路的改造设计所得到的电压测量结果很不精确，因为它使用电源作为参考电压。但是，可以将后侧LDR替换为合适取值的参考电压。如图11-6或图11-8所示，可以将该输入重新配置为参考电压输入。此时需要对软件缩放比例进行修改，反映出参考电压值的变化。通过相关修改，可以开发出较高精度的电压计。

11.8.4 其他测量系统

对于这里用到的一些想法，可以非常容易地应用到其他测量系统中。这对于任何具有电压输出的传感器都适用。例如，有很多半导体温度传感器都具有电压输出，输出值与温度直接相关。使用这种传感器(可以连接到飞线上，也可以焊接在原型区域中)，通过调整缩放比例，就可以方便地开发出数字温度计。

331

11.9 将Derbot配置为寻光机器人

要将Derbot配置为寻光机器人，需要在面板上添加LDR及其电阻。从本书附属资源中下载Dbt_light_seek项目，并围绕该程序创建MPLAB项目。根据具体应用修改程序初始化代码，将所有未用端口位设置为输出。编译连接程序并下载至微控制器。然后，Derbot可以立刻运行于寻光模式。在光强有清晰变化的环境下，Derbot可以很好地运行。但对于光线斑驳而有间隙的环境，它就会显得不知所措。如果周围光强均匀，它就会完全静止在原地。

小结

- 传感器所产生的多数信号本质上都是模拟的，但是微控制器能完成的所有处理活动都是数字的。
- 使用模数转换器(ADC)可以将模拟信号转换为数字形式。ADC通常只构成尺寸稍大的数据采集系统的一部分。ADC是一种联系模拟世界与数字世界的极为有效的接口。

- □ 在应用 ADC 和数据采集系统时需要相当小心，尤其需要注意时序要求、信号调理、接地以及参考电压的使用。
- □ 16F873A 具有 10 位 ADC 模块，其中包含有数据采集系统的特性。在应用该模块之前首先需要对这类系统加以了解。
- □ 一旦采集到数据值，就需要进一步加以处理，包括偏置、比例缩放和编码转换。这些任务有对应的标准算法，相应的汇编库已经发布。
- □ 比较器是联系模拟世界与数字世界的一种简单接口，通常用于对模拟信号进行分类，确定其属于两种状态中的哪一类。

参考文献

11.1. PICmicro® Mid-Range MCU Family Reference Manual (1997). Microchip Technology, Section 23, DS31023A.

11.2. Fixed Point Routines (1996). Microchip Technology, 671, DS00617B.

11.3. Watt-Hour Meter using PIC16C923 and CS5460. (2000). Microchip Technology, 671, DS00220A.

- □ ADC 和数据采集系统时需要相当小心，尤其需要注意时序要求、信号调理、接地以及参考电压的使用。
- □ 16F873A 具有 10 位 ADC 模块，其中包含有数据采集系统的特性。在应用该模块之前首先需要对这类系统加以了解。
- □ 一旦采集到数据，就需要进一步加以处理，包括偏置、比例缩放和编码转换。这些任务有对应的标准算法，相应的汇编程序已经发布。
- □ 比较器是模拟世界与数字世界的一种简单接口，通常用于对模拟信号进行分类，确定其属于两种状态中的哪一类。

参考文献

11.1. PICmicro® Mid-Range MCU Family Reference Manual (1997). Microchip Technology, Section 23, DS31023A.

11.2. Fixed Point Routines (1996). Microchip Technology, 671, DS00617B.

11.3. Watt-Hour Meter using PIC16C923 and CS5460. (2000). Microchip Technology, 671, DS00220A.

第四部分　更灵巧的系统与 PIC® 18FXX2

这一部分将转向介绍微控制器更为复杂的处理能力以及与之匹配的编程技术。其中将对 PIC 18 系列进行介绍，并将其作为基本硬件来使用。同时，这一部分的重点将主要放在软件部分：介绍 C 编程语言与使用方法；讲解嵌入式环境下的基本概念，如多任务和实时问题等；最后给出 Salvo™实时操作系统的应用实例。

第 12 章 更灵巧的系统与 PIC® 18FXX2

本书至此已经讲述了 PIC 16 系列中的 2 种微处理器，分别为小型和大型微处理器。它们都是很好的处理器，但在使用过程中仍然会受到一些约束和限制。其硬件设计在某些方面有很大的局限性，例如栈尺寸较小，并且所有中断源必须共用一个中断向量。就指令集而言，RISC 方式的优点值得肯定。但同时，缺少某些类型的指令同样让人感到无奈。例如，分支指令必须由 **skip** 和 **goto** 组合完成。并且，要使用所有这些零散的汇编指令构建主程序，这令人望之生畏。

人们之所以会时时感受到这些限制，是因为 16 系列始终保留着一颗仅为简单应用所设计的内核。如今，外围设备数量急剧增加，存储器容量也不断扩大。鉴于此，有必要建立新的起点来处理 16 系列的局限性问题，并重新思考微控制器内核的设计。PIC 18 系列正是基于这些原因而产生的。

在从 16F84A 开始逐步深入学习的过程中，我们发现 16F873A 保留了相同的处理器内核并增强了外围设备。而在这里，18 系列器件则改进了处理器内核，同时在很大程度上维持外围设备不变，从而创造出一种全新类型的微处理器。另外，PIC 结构最显著的优点——RISC 体系结构、较高的处理速度等，自然都得到了保留。然而，它所显示出的新特性则将 PIC 微控制器推向了更广阔的舞台，这些新特性增强了实时运行能力，便于高级语言的使用，并允许与更大容量的存储器进行交互。在 18 系列中，18F242 与 18FXX2 子型号有密切的关系，因而选择它进行深入学习。

本章期望读者能够以 16 系列器件为基础逐步过渡到 18 系列器件，因此在讲解时对二者进行了比较。从中我们将饶有趣味地了解设计理念是如何演进的。这种讲解的方式尤其体现在对指令集的描述中。

本书由此开始的后续章节将使用 C 语言进行编程，而不是汇编语言。因此，深入了解微控制器的细节就变得不那么重要了，C 编译器负责处理这些问题。18 系列器件本身并不简单，因此我们将从中感受到极大的解脱。基于上述原因，对于前几章（如第 7 章）已经介绍过的内容，本章将仅对某些主题进行“蜻蜓点水”式的介绍。

本章将首先对 18 系列和 18F242 进行简单的介绍，然后集中讲解其主要特性，主要是内核和存储器。将特别介绍以下内容：

- ☐ 18FXX2 类型微控制器的总体架构；
- ☐ 18 系列微控制器内核的结构与运行；
- ☐ 18 系列指令集；
- ☐ 存储器结构及其访问与寻址方法；

□ 18FXX2 的中断结构；
□ 18FXX2 的电源、复位与振荡器。

12.1 PIC 18 系列及 18FXX2 概述

PIC 18 系列微控制器对 PIC 内核进行了极大的增强，使其适用于高级的嵌入式项目。除了添加一些新特性之外，它还进行了某些专门设计从而能方便地从 16 系列器件向上移植。因此，正在完成这种转变的嵌入式系统设计员将会发现很多熟悉的东西。下面列出了一些主要特性。

1. 与 16 系列相似的地方

□ RISC(精简指令集计算机)、流水线结构、8 位 CPU，具有单个工作寄存器(W)与状态寄存器。
□ 很多外围设备相同或极为类似。
□ 类似的封装与引脚。
□ 很多特殊功能寄存器(SFR)和位名称未改变。
□ 16 系列指令中仅有 1 条未包含在 18 系列指令集中。
□ 指令周期由 4 个振荡器周期构成。

2. 18 系列的新特点

□ 指令数目达到 2 倍以上，具有 16 位指令字。
□ 增强的状态寄存器。
□ 8×8 硬件乘法器。
□ 更多的外部中断。
□ 两个优先级不同的中断向量。
□ 完全不同的存储器结构，更大的存储器容量。
□ 针对程序存储器与数据存储器的增强的地址产生方式。
□ 更大的栈，用户可以对其中的某些进行访问和控制。
□ 锁相环时钟发生器。

18FXX2 类型的微控制器包含 4 种密切相关的器件，它们的主要特性列在表 12-1 中。这张表格在很多方面与表 2-1 中的 16F87XA 微控制器相类似。所有 18FXX2 器件的指令集都由 75 条指令构成，时钟晶振可以工作于 DC 至 40 MHz。另外，每种微控制器都有对应的“低功耗”类型，编码为 18LFXX2。要了解该型号完整的厂商数据，可以查看参考文献 12.1。

图 12-1 描绘了 18FXX2 型号的双列直插式封装的引脚排列。该图与图 7-1 非常相似，端口、电源、振荡器和复位引脚都放置在同样的位置。这样，只需稍做更改就可以完成升级。

表 12-1 18FXX2 子型号

器件编号	引脚数目*	存 储 器	外围设备/特性
18F242	28	16 KB 程序存储器	3 个并行端口,4 个计数器/定时器
		(8K 指令**)	2 个捕捉/比较/PWM 模块
		768 字节 RAM	2 个串行通信模块
		256 字节 EEPROM	5 个 10 位 ADC 通道
18F252	28	32 KB 程序存储器	3 个并行端口,4 个计数器/定时器
		(16K 指令**)	2 个捕捉/比较/PWM 模块
		1536 字节 RAM	2 个串行通信模块
		256 字节 EEPROM	5 个 10 位 ADC 通道
18F442	40	16 KB 程序存储器	5 个并行端口,4 个计数器/定时器
		(8K 指令**)	2 个捕捉/比较/PWM 模块
		768 字节 RAM	2 个串行通信模块
		256 字节 EEPROM	8 个 10 位 ADC 通道
18F452	40	32 KB 程序存储器	5 个并行端口,4 个计数器/定时器
		(16K 指令**)	2 个捕捉/比较/PWM 模块
		1536 字节 RAM	2 个串行通信模块
		256 字节 EEPROM	8 个 10 位 ADC 通道

* 仅适用于 DIP 封装

** 单字指令,注意某些是双字

ADC:模数转换器;PWM:脉宽调制

12.2 18F2X2 结构图与状态寄存器

图 12-1a 中所示的 28 个引脚的器件 18F2X2 微控制器所对应的框图如图 12-2 所示。仔细分析此框图,可以发现器件的主要特点。

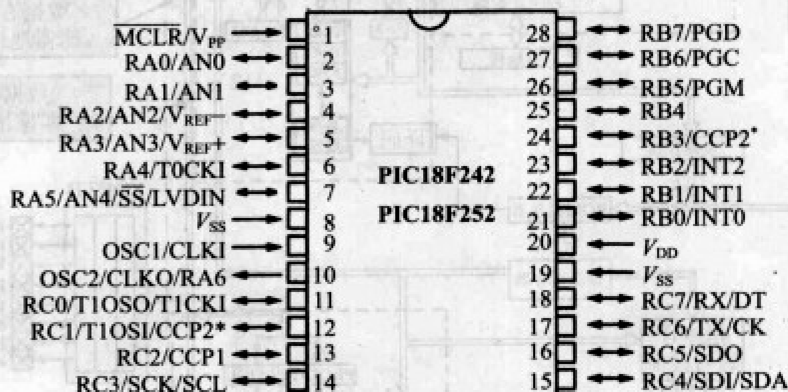
处在图中央位置的是 CPU(中央处理单元),它包含有 8 位 ALU(算术逻辑单元)、工作寄存器“WREG”(有时也称作累加器)和 8 位×8 位硬件乘法单元。程序存储器中的指令通过指令寄存器进行传递,进而决定 CPU 的动作。在 CPU 的上方和靠近左边的位置可以观察到这个过程。此外,CPU 还具有一个重要元素:状态寄存器,它在图中未显示。

图中左上角是程序存储器。地址总线进入存储器的“地址锁存器(Address Latch)”中。21 位的地址总线可以寻址 2^{21} 个地址,也即 2097152 个地址(2M 字节)。表 12-1 显示 18F242 可以寻址 16K 个地址,只需 14 位。因此,其他地址线在这里就是多余的。从“程序存储器数据锁存器”出来的 16 位总线用于传送指令字。在它的所有

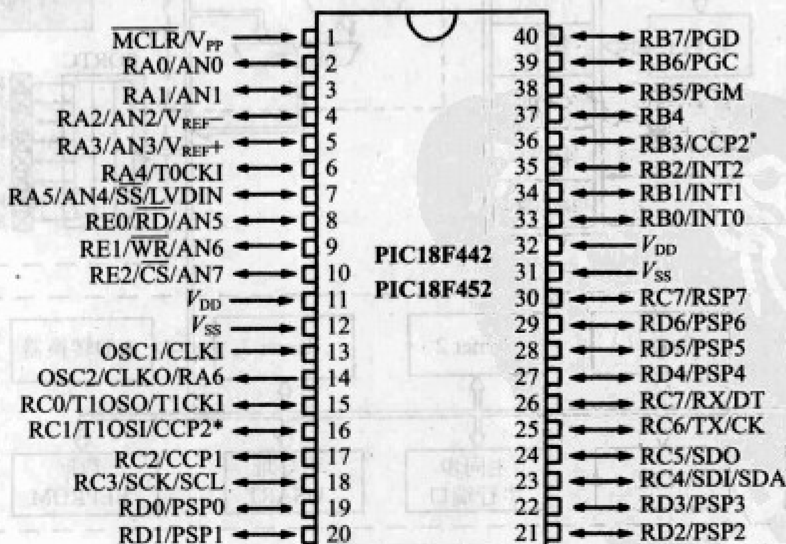
后续分支中，重点是将指令字送入上面提到的指令寄存器中。在程序存储器右侧有一块区域标记为"程序存储器地址产生"。其核心自然就是程序计数器(Program Counter)。程序计数器下方是栈(Stack)，包含31个单元。表指针(Table Pointer)可以在用户程序控制下对表或程序存储器中的其他数据进行访问。

337

位于图的上中部的是数据存储器。与程序存储器类似，它的地址产生部分构成了概览图中的一个重要模块，其中包括文件选择寄存器(如FSR0)和存储区选择寄存器(Bank Select Register，BSR)。数据存储器地址为12位，可寻址4096B。表12-1显示该地址总线并未被完全利用。从数据存储器进出的数据都经过主数据总线。



(a) 18F242和18F252



(b) 18F442和18F452

*RB3与CCP2复用同一个引脚

图12-1 PIC 18FXX2引脚图和DIL封装

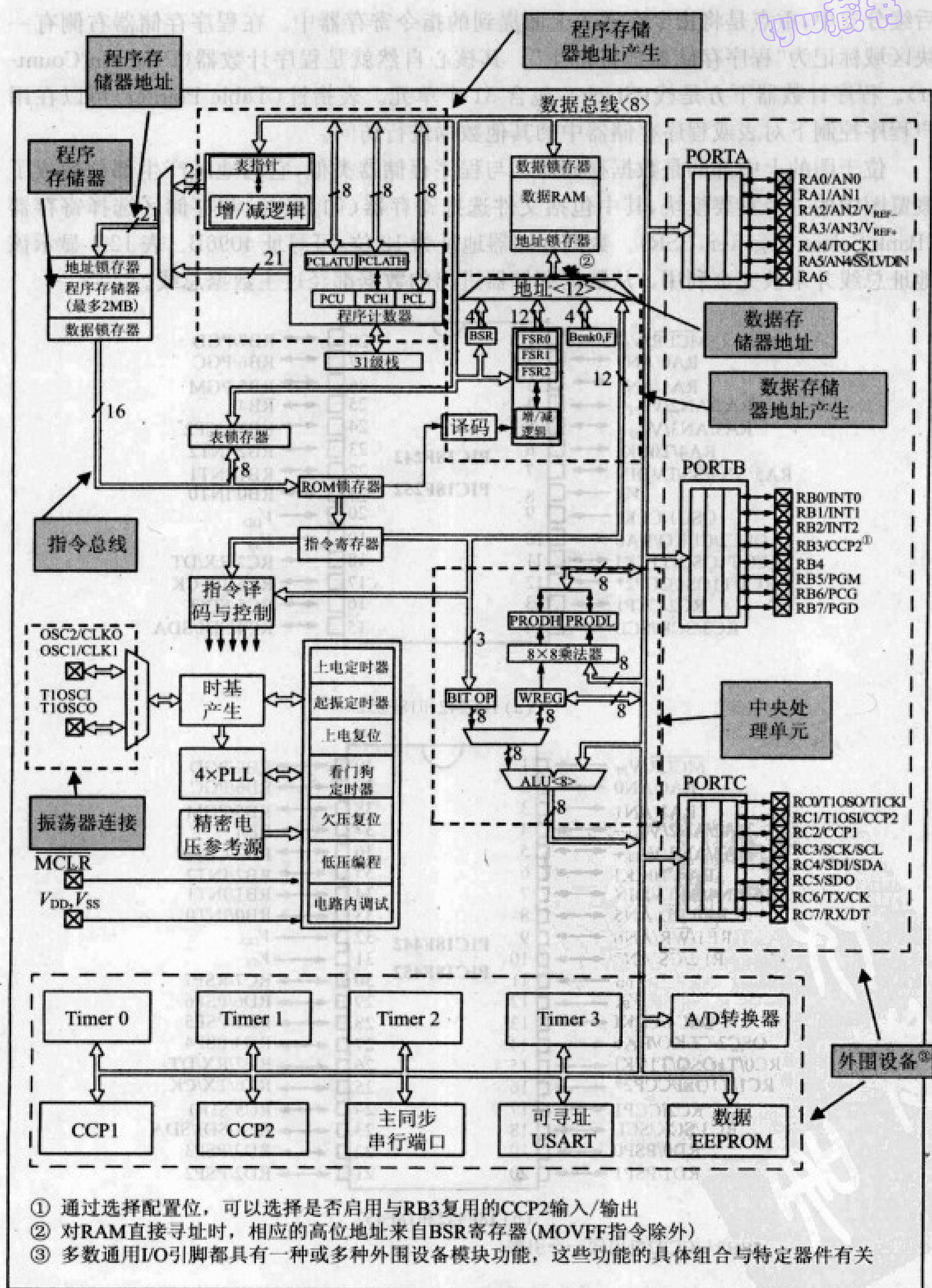


图 12-2　PIC 18F2X2 框图(阴影框中所附标签为作者所加)

对于程序存储器与数据存储器，它们具有相互分离的地址总线和数据输入/输出，此时可以确信微控制器内部为哈佛结构。

338～339

图中靠近左下角的位置是微控制器的电源连接 V_{DD} 和 V_{SS}。观察图 12-1 可以发现，图中的两种封装中都有 2 个专用引脚与 V_{SS} 连接，这样可以确保良好的 0V 连接。其中较小的封装具有单个 V_{DD} 连接，而较大的封装则具有 2 个这样的连接。与电源相关的部分有上电复位（Power-on Reset）、上电定时器（Power-up Timer）和欠压复位（Brown-out Reset）。上电复位可以确保在电源开启时进行复位操作，上电定时器用于在上电之后将微控制器维持在复位状态并保持一段固定的时间，欠压复位用于在掉电情况下将微控制器强制复位。

在图的左方可以看到振荡器输入，包括主振荡器和定时器 1 的振荡器输入。与主振荡器相连的是时基产生电路，由此可以获得不同的内部时钟信号。同样地，与主振荡器相关的是起振定时器（Oscillator Start-up Timer），使用它可以确保在微控制器自身开始运行之前振荡器能够稳定可靠地运行。新的振荡器元件是锁相环（phase-locked loop，PLL）。利用它可以选择引入与振荡器频率高数倍的时钟，从而使整个运行速度加快，但会对运行的稳定性和功耗产生影响。

在图的左下区域可以看到看门狗定时器（Watchdog Timer）和电路内调试器（In-Circuit Debugger）。前者是一种定时元件，用于在程序意外崩溃时使微控制器强制复位。后者则用于提供一种切实设计在微控制器内部的诊断能力。要利用这一功能，需要首先将微控制器与宿主计算机相连，然后使用该工具在计算机上控制程序的执行。此时，还可以将诊断数据（包括微控制器寄存器的值）返回到计算机上显示出来。在第 7 章的 7.11 节已经对这一特性进行了描述。

最后，在图的右边放置了并行端口，包括它们的输入/输出功能以及与外围设备的所有连接。其他外围设备则放置在图的下部，其中还包含一个使用 EEPROM 技术的存储器模块。

图 12-3 所示的是状态寄存器，它是 CPU 的重要组成部分。其中包含有完整的 5 位状态信息，用于描述微控制器最近执行的操作状态。相形之下，16 系列微控制器有限的状态位（见图 7-3）则成为它的缺点之一。18 系列添加了 2 个新的状态位：OV（位 3）和 N。OV 用于指示 8 位范围的溢出，N 用于指示某个二进制补码数为负。由于二进制补码数的符号位就是它的 MSB，因此状态位 N 就是结果的 MSB。利用这些附加位可以改进程序分支并获得更好的算术能力。另外，图中还对其他信息给出了清晰的解释。

U-0	U-0	U-0	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
—	—	—	N	OV	Z	DC	C
位 7							位 0

位 7~位5 未实现：读作“0”

位 4 N：负数标志位
此位用于有符号的算术运算（二进制补码），
表明结果是否为负（ALU MSB=1）
1=结果为负
0=结果为正

位 3 OV：溢出标志位
此位用于有符号算术运算（二进制补码），表明7位数量级的溢出，溢出将导致符号位（位 7）改变状态
1=有符号算术运算发生溢出（在本次运算中）
0=未发生溢出

位 2 Z：零标志位
1=算术或逻辑运算结果为0
0=算术或逻辑运算结果不为0

位 1 DC：数字进/$\overline{借}$位
用于ADDWF、ADDLW、SUBLW和SUBWF指令
1=结果的第4低位发生了进位
0=结果的第4低位未发生进位
注：$\overline{借}$位的极性是相反的，减法运算通过加上第2个操作数的二进制补码来实现。对于移位指令（RRF、RLF），此位值来自于源寄存器的位 4或位 3

位 0 C：进/$\overline{借}$位
用于ADDWF、ADDLW、SUBLW和SUBWF指令
1=结果的最高位发生了进位
0=结果的最高位未发生进位
注：$\overline{借}$位的极性是相反的，减法运算通过加上第2个操作数的二进制补码来实现。对于移位指令（RRF、RLF），此位值来自于源寄存器的最高位或最低位

图 12-3 PIC 18FXX2 状态寄存器

12.3 18 系列指令集

PIC 18 系列的完整指令集显示在附录 5 中，包含有 75 条不同指令。它是由 PIC 16 系列和 PIC 17 系列指令集所构成的超集，因此任何能够在这两类微控制器上运行的程序都可以运行于 18 系列上。如此众多的指令使人感觉它更像 CISC（复杂指令集计算机，参见第 1 章），而丧失了 RISC 所带来的简单性。另外注意，还有一种“扩展的”18 系列指令集，它向其中又添加了若干指令。这在第 13 章中进行了描述。

查看表 A5-1，要了解所有这些指令似乎不是一件轻松的事。幸运的是，我们无需了解它们！在本章之后，几乎所有程序都将用 C 语言来编写，因此无需担心汇编语言。然而，大概地了解它们仍然是有用的，毕竟有时需要在 C 程序中插入汇编程序。

如果要从 16 系列转向 18 系列，可以从表 12-2 中获得有趣的体验。它对两种指令

340

集进行了简要的比较。表中第1列列出了所有的16系列指令,与附录1所给出的顺序基本相同。第2列则给出了等价的18系列指令(如果有的话),并列出了所有全新的指令。

表 12-2 16系列与18系列指令集的比较

16系列指令	18系列等价指令	指令描述
面向字节的文件寄存器操作		
addwf f,d	addwf f,d,a	W与f相加
	addwfc f,d,a	W与f及进位相加
andwf f,d	andwf f,d,a	W与f相加
clrf f	clrf f,a	f清零
clrw	—	W清零
comf f,d	comf f,d,a	对f取反
—	cpfseq f,a	比较f与W,相等则跳过
—	cpfsgt f,a	比较f与W,f大则跳过
—	cpfslt f,a	比较f与W,f小则跳过
decf f,d	decf f,d,a	f减1
decfsz f,d	decfsz f,d,a	f减1,为0则跳过
—	decfsnz f,d,a	f减1,非0则跳过
incf f,d	incf f,d,a	f加1
incfsz f,d	incfsz f,d,a	f加1,为0则跳过
	incfsnz f,d,a	f加1,非0则跳过
iorwf f,d	iorwf f,d,a	f与W进行"同或"运算
movf f,d	movf f,d,a	传送f
—	movff f_s,f_d	传送源文件f_s至目标文件f_d
movwf f	movwf f,a	传送W至f
nop	nop	无操作——该指令用于实现某些特定意图
	nop	双字指令的第2个字,为了避免被意外地当成一条指令来执行,因而将其编码为nop
—	mulwf f,a	W与f相乘
—	negf f,a	对f求补
rlf f,d	rlfc f,d,a	带进位循环左移f
	rlnfc f,d,a	不带进位循环左移f
rrf f,d	rrfc f,d,a	带进位循环右移f
	rrncf f,d,a	不带进位循环右移f
—	set f	置位f
subwf f,d	subwf f,d,a	f减去W
	subwfb f,d,a	f减去W和借位
—	subfwb f,d,a	W减去f和借位

（续）

16 系列指令	18 系列等价指令	指令描述
swapf f,d	swapf f,d,a	f 半字节交换
—	tstfsz f,a	测试 f,为 0 则跳过
xorwf f,d	xorwf f,d,a	W 和 f 进行"异或"操作
面向位的文件寄存器操作		
bcf f,b	bcf f,b,a	寄存器 f 的 b 位清零
bsf f,b	bsf f,b,a	寄存器 f 的 b 位置位
—	btg f,d,a	寄存器 f 的 b 位取反
btfsc f,b	btfsc f,b,a	测试 f 中的 b 位,为 0 则跳过
btfss f,b	btfss f,b,a	测试 f 中的 b 位,为 1 则跳过
立即数操作		
addlw k	addlw k	W 与立即数相加
andlw k	andlw k	W 与立即数进行"与"操作
iorlw k	iorlw k	W 与立即数进行"或"操作
movlw k	movlw k	将立即数发送至 W
—	movlb	将立即数发送至 BSR
—	lfsr f,k	将 12 位立即数 k 装载至 FSR f
—	mullw	W 与立即数相乘
sublw k	sublw k	立即数减去 W
xorlw k	xorlw k	W 与立即数进行"异或"操作
控制操作		
call k	call n,s	调用子例程,s=1 保存上下文至栈,s=0 则不保存
	rcall n	相对调用子例程
clrwdt	clrwdt	看门狗定时器清零
—	daw	十进制调整 W
goto k	goto n	转移至绝对地址 k/n,可以是程序存储器空间中的任何地方
—	pop	将返回栈顶部的内容弹出(TOS)
—	push	将内容压入返回栈的顶部(TOS)
—	reset	软件复位
retfie	retfie s	由中断返回,s=1 从栈找回上下文,s=0 则不找回
retlw k	retlw k	返回时将立即数送至 W
return	return s	由子例程返回,s=1 从栈找回上下文,s=0 则不找回
sleep	sleep	进入休眠模式

341～342

（续）

16系列指令	18系列等价指令	指令描述
—	bc, bn, bnc, bnn, bnov, bnz, bov, bz	8个条件分支指令，每条指令对应状态寄存器位N、OV、Z、C的一种状态，都具有8位二进制补码相对地址n
—	bra n	无条件转移到8位二进制补码相对地址n
程序存储器读/写表指令		
—	tblrd *, tblrd * +, tblrd * −, tblrd+ *	4种读表指令，指针变化方式分别为不变、读表后增1、读表后减1、增1后读表
—	tblwt *, tblwt * +, tblwt * −, tblwt+ *	4种写表指令，指针变化方式分别为不变、写表后增1、写表后减1、增1后写表

在深入了解18系列指令之前，首先来看一下表A5-1的前2列。它们分别给出了汇编指令助记符和操作数，以及指令的功能。A5-2对其中所用的操作数符号进行了解释，如a、d、f。其中有一些熟悉的操作符来自于16系列，并加入了新的位操作数a，使得指令可以具备3个操作数。操作数a用来定义存储器区域“存取RAM(Access RAM)”，本章后续部分将对它进行描述。现在，程序员可以通过a来选择是否使用存取RAM。

表A5-1中的第3列用于描述执行指令所需的指令周期数。由于采用了RISC和流水线结构，所有正常指令可以在单个周期内执行。能够导致程序分支的指令需要2个周期。跳转(skip)指令如果未发生跳转则只需1个周期，如果后面紧跟着一个单字指令则需要2个周期，如果紧跟着一个双字指令则需要3个周期。此外，还有少数复杂指令的执行时间大于1个周期。

343

表中第4列给出了指令的实际编码。幸运的是，这种编码是由汇编器或编译器产生的。大多数指令只占用一个16位的单字，只有4条指令占用了2个16位字。它们是：call、goto、movff和lfsr。仔细观察这些指令各自的机器码，可以发现一个非常有趣的现象，虽然每条指令的第2个字中都包含了有用的信息，但是如果单独来看则都编码为nop指令。因为，只要最高4位符合nop指令，就将其作为nop的机器码，而不管低位的具体内容是什么。这样安排的原因在于，任何时候如果微控制器试图将该第2个字解释为一条独立的指令，程序都将重新对齐执行位置。

表A5-1中的最后一列为“受影响的状态”，标明了指令执行过程中将影响状态寄存器(见图12-3)的那些位。从这个角度出发可以对那些同时出现在16系列指令集(附录1)和18系列指令集中的“相同”指令进行比较。例如，16系列中的addwf只影响C、DC和Z位。而在18系列中，不仅影响这些位，还影响到OV和N。虽然指令功能没有改变，但通过改变更多的状态位可以获得更强大的编程能力。

观察表12-2，与16系列指令进行比较，可以将18系列指令归为下列几种类型。

12.3.1 未变化的指令

这类指令的功能和形式与16系列相同。在立即数指令中有很多这样的例子，如addlw k，andlw k。唯一不同的是受影响的状态位个数发生了变化。

12.3.2　经过升级的指令

这类指令与16系列中的原有指令几乎相同，只是添加了功能和灵活性，部分原因是体系结构发生了变化。其中最直接的例子就是上面提到的存取RAM，编程中可以选择将其作为目标存储器区域。在面向字节的算术和逻辑操作指令中，可以找到很多这样的例子，如addwf f,d,a和andwf f,d,a。

另一个有趣的变化是在使用call、return和retfie指令时的灵活性。使用新的操作数s可以选择是否将上下文保存至栈，以及是否从栈中提取出来。

为保证向上兼容性，所有这些指令都将汇编为有效的18系列代码，即便是以16系列格式表示，也依然如此。

344

12.3.3　变化而来的新指令

某些16系列指令存在局限性，要在某些情况下更有效地使用它们，就需要将其与其他指令联合使用。18系列指令集对很多这样的指令进行了更改，从而突破这些限制。例如，对于简单的加指令addwf，现在可以进行带进位的加法，即addwfc。这样就简化了大量16位或更高位的加法。类似地，还可以进行带借位的减法、带进位或不带进位的移位，以及incfsnz(f增1，若非0则跳过)。

12.3.4　全新指令

最后，18系列中还包括很多全新的指令。它们的产生大多来源于硬件或存储器寻址技术的增强。在算术指令中，可以使用mulwf(W和f相乘)和mullw(W和立即数相乘)来实现乘法这样的重要运算。它们使用了硬件乘法器，这在图12-2中已有描述。乘数和被乘数都被视为无符号数，其结果放在寄存器PRODH和PRODL中。值得注意的是，乘法指令不改变状态标志位，即便存在结果为0的情况，也依然如此。

指令集中还添加了其他一些重要指令：表的读写指令、数据传送至栈或从栈中取出的指令，以及更多的条件分支指令。它们建立在状态寄存器中新增的条件标志位的基础之上。另外利用某些指令可以方便地产生条件分支，这类指令包括一系列的比较指令(如cpfseq)和测试指令tstfsz。

传送指令movff是一个有用的新指令，它可以直接将一个存储器地址的内容传送至另一个地址。注意这条指令代码占用2个字，需要2个周期来执行。因此，相对于它所替代的2条16系列的指令而言，优势也许并不明显。但是，它确实能够避免重写W寄存器的内容。

在12.10节的例程与练习中，将对其中的一些新指令进行深入学习。

12.4 数据存储器与特殊功能寄存器

表 12-1 列出了 18FXX2 型号中各种器件的不同存储器的大小。下面这一节中，在需要考察存储器的地方将使用 18F242 作为主要的例子。型号中的其他器件与此类似，同时又都具有更大的存储器容量。如果您是从 PIC 16 系列开始学习的，那么现在也许需要从新的角度来认识存储器结构。

12.4.1 数据存储器映射

图 12-4 显示了 18F242 的数据存储器映射概要图。每个存储器单元(location)占用 1 个字节，12 位地址最多可以寻址 4096 个单元。存储器结构被有效地分成 16 个存储区(bank)，每个存储区占用 256 个字节。一种特殊的寄存器 BSR(Bank Select Register，存储区选择寄存器)中保存的数据位负责对存储区进行选择。在图 12-4 中可以看到这些位，它们构成了 12 位存储器地址的高 4 位。图中，18F242 只有最低的 3 个块被实现为“通用寄存器(general-purpose registers，也即通用 RAM)”，它们构成了表 12-1 所示的 768(3×256)个字节的数据存储器。

345

数据存储器低端的 3 个存储区被用作通用 RAM；同时，SFR 则包含在存储器高端的存储区中，位于该高端存储区的上半部分。这显示在图 12-5 中。注意，该图中的 4 列并非是存储器的存储区。事实上，由图 12-4 已经看出，这 4 列合在一起只是构成了最高存储区的一部分。

12.4.2 存取 RAM

在前面浏览指令集时已经偶然提到了“存取 RAM(Access RAM)”。在图 12-4 中有 2 个存储器区域附带有这样的名称。它们分别是存储器的最低 128 字节和最高 128 字节。处于最低位的是通用 RAM，而处于最高位的则包含了所有 SFR。存取 RAM 的概念提供了一种快速访问部分 RAM 的方式。虽然存取 RAM 的两部分分别处于存储器映射中相反的两端，但在用作存取 RAM 时则被看作一个连续的存储区。此时，BSR 的位值将被忽略，只具有 8 位地址，SFR 地址紧跟在低端存取 RAM 地址之后。

如表 12-2 或附录 5 所示，存取 RAM 可用于所有具有 a 操作数的指令。如果 a 被置为 0，就只能使用存取 RAM，并按照上述方法进行访问。

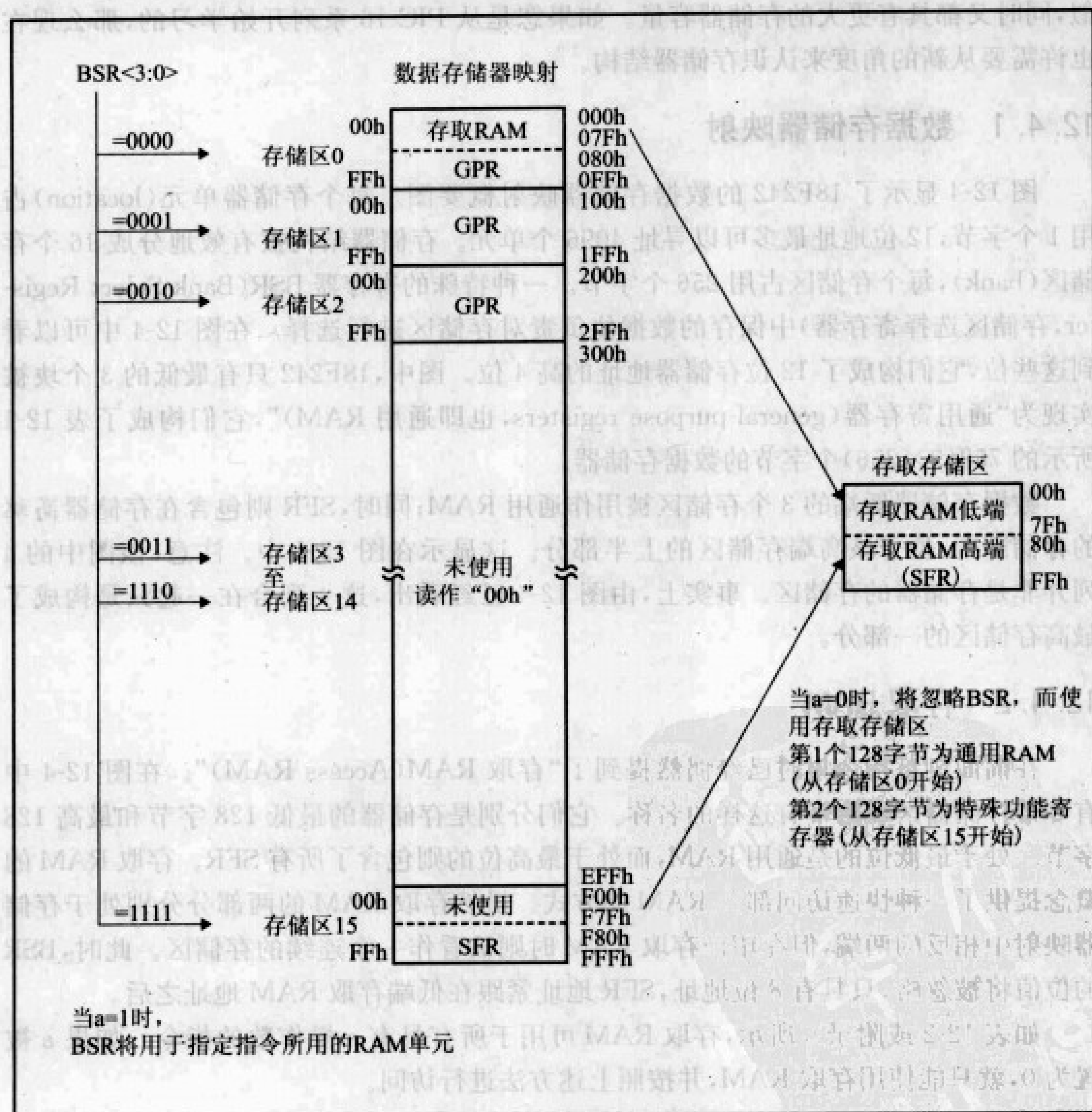
12.4.3 间接寻址以及在数据存储器中访问表格

第 5 章的 5.4.1 节已经介绍了 16 系列中的间接寻址概念。在间接寻址中，使用文件选择寄存器(File Select Register，FSR)来保存某个地址。如果程序对(不存在的)寄存器 INDF 进行寻址，存放在 FSR 中的地址将被用作指令所需的地址。

上述概念对 18 系列依然适用，但扩展为多个 FSR 和 INDF 寄存器，从而与更大的

存储器容量相匹配。首先，FSR 寄存器已增为 3 个（如图 12-2 所示）。由于存储器容量的提高，每个 FSR 都需要占用 12 位。因此，每个 FSR 都由 2 个存储单元构成，在图 12-5 所示的 SFR 映射中分别标记为 FSR2H：FSR2L、FSR1H：FSR1L 和 FSR0H：FSR0L。

为应对更为复杂的情况，为 INDF 寄存器提供了 5 个等价寄存器。表 12-3 列出了这些寄存器，在图 12-5 的 SFR 映射中也可以找到。



对照表：
BSR：Bank Select Register，存储区选择寄存器
GPR：General-purpose Register，通用寄存器
SFR：Special Function Register，特殊功能寄存器

图 12-4 PIC 18F242 数据存储器映射

地址	名称	地址	名称	地址	名称	地址	名称
FFFh	TOSU	FDFh	INDF2③	FBFh	CCPR1H	F9Fh	IPR1
FFEh	TOSH	FDEh	POSTINC2③	FBEh	CCPR1L	F9Eh	PIR1
FFDh	TOSL	FDDh	POSTDEC2③	FBDh	CCP1CON	F9Dh	PIE1
FFCh	STKPTR	FDCh	PREINC2③	FBCh	CCPR2H	F9Ch	—
FFBh	PCLATU	FDBh	PLUSW2③	FBBh	CCPR2L	F9Bh	—
FFAh	PCLATH	FDAh	FSR2H	FBAh	CCP2CON	F9Ah	—
FF9h	PCL	FD9h	FSR2L	FB9h	—	F99h	—
FF8h	TBLPTRU	FD8h	STATUS	FB8h	—	F98h	—
FF7h	TBLPTRH	FD7h	TMR0H	FB7h	—	F97h	—
FF6h	TBLPTRL	FD6h	TMR0L	FB6h	—	F96h	TRISE②
FF5h	TABLAT	FD5h	T0CON	FB5h	—	F95h	TRISD②
FF4h	PRODH	FD4h	—①	FB4h	—	F94h	TRISC
FF3h	PRODL	FD3h	OSCCON	FB3h	TMR3H	F93h	TRISB
FF2h	INTCON	FD2h	LVDCON	FB2h	TMR3L	F92h	TRISA
FF1h	INTCON2	FD1h	WDTCON	FB1h	T3OON	F91h	—
FF0h	INTCON3	FD0h	RCON	FB0h	—	F90h	—
FEFh	INDF0③	FCFh	TMR1H	FAFh	SPBRG	F8Fh	—
FEEh	POSTINC0③	FCEh	TMR1L	FAEh	RCREG	F8Eh	—
FEDh	POSTDEC0③	FCDh	T1CON	FADh	TXREG	F8Dh	LATE②
FECh	PREINC0③	FCCh	TMR2	FACh	TXSTA	F8Ch	LATD②
FEBh	PLUSW0③	FCBh	PR2	FABh	RCSTA	F8Bh	LATC
FEAh	FSR0H	FCAh	T2CON	FAAh	—	F8Ah	LATB
FE9h	FSR0L	FC9h	SSPBUF	FA9h	EEADR	F89h	LATA
FE8h	WREG	FC8h	SSPADD	FA8h	EEDATA	F88h	—
FE7h	INDF1③	FC7h	SSPSTAT	FA7h	EECON2	F87h	—
FE6h	POSTINC1③	FC6h	SSPCON1	FA6h	EECON1	F86h	—
FE5h	POSTDEC1③	FC5h	SSPCON2	FA5h	—	F85h	—
FE4h	PREINC1③	FC4h	ADRESH	FA4h	—	F84h	PORTE②
FE3h	PLUSW1③	FC3h	ADRESL	FA3h	—	F83h	PORTD②
FE2h	FSR1H	FC2h	ADCON0	FA2h	IPR2	F82h	PORTC
FE1h	FSR1L	FC1h	ADCON1	FA1h	PIR2	F81h	PORTB
FE0h	BSR	FC0h	—	FA0h	PIE2	F80h	PORTA

① 未实现寄存器读作“0”
② 该寄存器在PIC 18F2X2器件中不可用
③ 不是实际存在的寄存器

图 12-5 PIC 18F242 特殊功能寄存器

如果程序指令对表 12-3 中的任何单元进行寻址，那么存放在相应 FSR 中的 12 位数将作为间接地址供指令使用，指令将访问数据存储器中的相应地址并进行操作。在指令执行过程中，FSR 的值将按照表 12-3 所示进行修改。

表 12-3 间接寻址中使用的"虚拟"寄存器

被寻址的"虚拟"寄存器 n=0、1 或 2	使用 FSR 寻址的指令的后续动作
INDFn	FSRn 无变化
POSTINCn	在访问之后自动递增 FSR 的值
POSTDECn	在访问之后自动递减 FSR 的值
PREINCn	在访问之前自动递增 FSR 的值
PLUSWn	WREG 中的值与 FSRn 相加作为间接地址。FSR 与 WREG 的值都不发生改变

12.5 程序存储器

346~347

与上述数据存储器类似，下面将以 18FXX2 中的最小器件 18F242 为例对程序存储器进行分析。

12.5.1 程序存储器映射

程序存储器映射如图 12-6 所示。从图中可以看出，它只占用了 21 位总线所能提供的整个存储空间中的一小部分。每个存储器单元为 1 个字节。常规指令字占用 16 位，因此每条指令占用 2 个或 4 个字节。低位字节存储在偶地址所在单元中。复位向量（程序执行起始点）在存储器单元 0000 中，另外有 2 个中断向量位于单元 0008_H 和 0018_H 中。在编写中断服务程序（Interrupt Service Routine，ISR）时，应将程序起始位置设置为这两个单元之一。

12.5.2 程序计数器

图 12-6 顶端的程序计数器（Program Counter）的宽度为 21 位。由于每条指令占据 2 个或 4 个字节，因此每执行 1 条指令，程序计数器将增加 2 或 4。

在存储器映射中，可以通过 SFR 来访问程序计数器。其中的低位字节被称为 PCL。它是一个可读可写的寄存器，在图 12-5 中位于存储器单元 $FF9_H$。程序计数器中位于中间的 1 个字节以及更高的 5 位不能直接进行读写，但可以通过寄存器 PCLATH 和 PCLATU（如图 12-2 和图 12-5 所示）进行更新。通过写 PCL 的操作，可以将这些单元的内容传送至程序计数器。类似地，通过读 PCL 的操作，可以将程序计数器中的相应高位传送至 PCLATH 和 PCLATU。

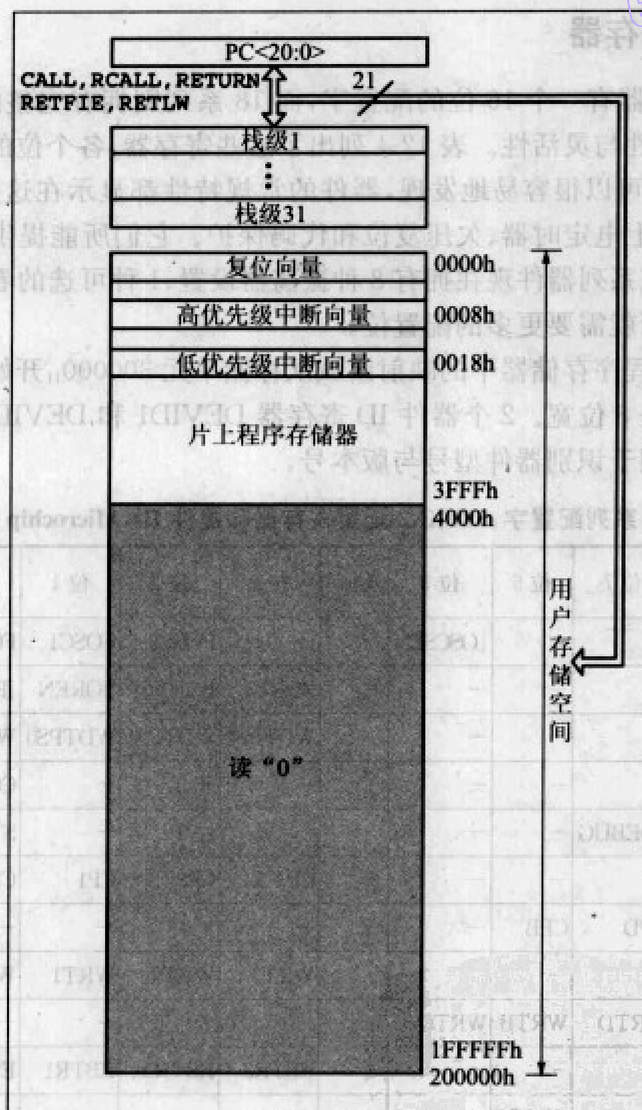


图 12-6 PIC 18F242 程序存储器映射和返回地址栈

12.5.3 在 16 系列基础上增强的计算 goto 指令

在 16 系列中,可以使用计算 goto(computed goto)的方法从查找表中提取数据,如图 5-5 或例程 5-4 所示。通过向程序计数器中加上偏移量可以跳转至一列 retlw 指令中的一个,然后该指令将导致返回主程序,同时 W 寄存器中将保留被选数据字节。这种方法在 18 系列中同样可行,但是特别要注意的是,现在每条指令占用程序存储器中的 2 个字节。因此,在 16 系列程序中向程序计数器添加的偏移量必须翻倍,才能在 18 系列程序中产生同样的结果。参考文献 12.2 给出了如何操作的示例。

348～349

12.5.4 配置寄存器

16 系列微控制器有一个 16 位的配置字，而 18 系列则拥有完整的配置寄存器组，显示出器件的复杂性与灵活性。表 12-4 列出了这些寄存器，各个位的功能完整地显示在表 12-5 中。从中可以很容易地发现，器件的常规特性都显示在这里，包括振荡器设置、看门狗定时器、上电定时器、欠压复位和代码保护。它们所能提供的选项数大多都有了增长。例如，该系列器件现在拥有 8 种振荡器设置、1 种可选的看门狗模式以及更多的欠压设置，因而就需要更多的配置位。

配置寄存器在程序存储器中的映射区域从存储单元 300000_H 开始。与程序存储器类似，这些单元都是 8 位宽。2 个器件 ID 寄存器 DEVID1 和 DEVID2 是只读寄存器，包含预编程信息，用于识别器件型号与版本号。

350

表 12-4 18 系列配置字：18FXX2 配置寄存器和器件 ID(Microchip 公司供表)

文件名		位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0	默认/未编程值
300001h	CONFIG1H	—	—	$\overline{OSCSEN}$	—	—	FOSC2	FOSC1	FOSC0	-1--111
300002h	CONFIG2L	—	—	—	—	BORV1	BORV0	BOREN	$\overline{PWRTEN}$	----1111
300003h	CONFIG2H	—	—	—	—	WDTPS2	WDTPS1	WDTPS0	WDTEN	----1111
300005h	CONFIG3H	—	—	—	—	—	—	—	CCP2MX	-------1
300006h	CONFIG4L	$\overline{DEBUG}$	—	—	—	—	LVP	—	STVREN	1----1-1
300008h	CONFIG5L	—	—	—	—	CP3	CP2	CP1	CP0	----1111
300009h	CONFIG5H	CPD	CPB	—	—	—	—	—	—	11------
30000Ah	CONFIG6L	—	—	—	—	WRT3	WRT2	WRT1	WRT0	----1111
30000Bh	CONFIG6H	WRTD	WRTB	WRTC	—	—	—	—	—	111-----
30000Ch	CONFIG7L	—	—	—	—	EBTR3	EBTR2	EBTR1	EBTR0	----1111
30000Dh	CONFIG7H	—	EBTRB	—	—	—	—	—	—	-1------
3FFFFEh	DEVID1	DEV2	DEV1	DEV0	REV4	REV3	REV2	REV1	REV0	(1)
3FFFFFh	DEVID2	DEV10	DEV9	DEV8	DEV7	DEV6	DEV5	DEV4	DEV3	0000 0100

表 12-5 18 系列配置字：配置位概要与器件 ID

配置位	功能概要[未编程值为 1，激活(启用)值为 0]
$\overline{OSCSEN}$	时钟源切换启用位
FOSC2：FOSC0	选择 8 种振荡器模式中的 1 种(参见表 12-6)
BORV1，BORV0	选择欠压复位电压，分别为 2.5，2.7，4.2，4.5V
BOREN	欠压复位启用位
$\overline{PWRTEN}$	上电定时器启用位

(续)

配 置 位	功能概要[未编程值为 1,激活(启用)值为 0]
WDTPS2 : WDTPS0	看门狗定时器后分频选择位,8 种取值从 1 : 1 至 1 : 128
WDTEN	看门狗定时器启用位
CCP2MX	CCP2 复用位,选择 RC1(1)或 RB3(0)
DEBUG	后台调试启用位
LVP	低压编程启用位
STVREN	栈满/下溢复位启用位
CP3 : CP0	代码保护位
CPD	数据 EEPROM 代码保护位
CPB	引导地址块代码保护位
WRT3 : WRT0	程序存储器写保护位
WRTD	数据 EEPROM 写保护位
WRTB	引导地址块写保护位
WRTC	配置寄存器写保护位
EBTR3 : EBTR0	读表保护位
EBTRB	引导地址块读表保护位
DEV2 : DEV0	器件 ID 位:000=18F252,001=18F452,100=18F242,101=18F442
REV4 : REV0	版本 ID 位
DEV10 : DEV3	补充器件 ID 位

351

12.6 栈

在高级微处理器中,栈是一种多用途的存储区域。有时栈的使用是自动进行的,例如保存子例程调用中的返回地址;另外程序员也可以利用栈来实现短期的数据存储。事实上,在很多情况下会需要使用多个栈。但是,正如前面所看到的,16 系列栈尺寸较小,且结构固定又不够灵活。它只有 8 层,并且直接与程序计数器相关,只是在某些子例程调用或中断条件下对 PC 值进行保存或返回。

18 系列栈在某种程度上与较大型微处理器的特性接近。它不仅为子例程调用等任务保留了自动的栈功能,同时也为用户提供了访问栈的能力。此外,在别的存储区域中还提供了小容量栈用于保存关键的数据寄存器。

12.6.1 自动栈操作

18 系列中的主栈被称为"返回地址栈(Return Address Stack)",用于和那些现有的其他尺寸较小的栈单元相区分。在图 12-6 中可以看到这个栈。它由 31 个可读写的存

储器单元构成，每个单元宽 21 位。图中还列出了那些能够引起栈自动操作的汇编指令助记符。除了 retfie(由中断指令返回)之外，这些指令都与子例程的调用或返回有关。

12.6.2 程序员对栈的访问

栈指针(Stack Pointer)未在图 12-6 中画出，它用于保存当前栈地址。它的值在任何复位情况下都会被置为 0，并且可以被所有的自动栈操作所改变。因此，它的值将在有数值压入栈时递增，并在有数值从栈中弹出时递减。同时，它还被配置为特殊功能寄存器 STKPTR 的一部分。与所有的 SFR 一样，程序员可以对它进行读写。在图 12-5中可以看到它位于存储单元 FFC_H。STKPTR 中的低 5 位是栈指针。另外，该寄存器中还包含有栈上溢出或下溢出的标志位，分别为 STKOVF(也称 STKFUL，位 7)和 STKUNF(位 6)。

程序员可以对栈顶单元所保存的内容进行读写。由于具有 21 位的宽度，它占用 3 个寄存器单元：TOSU、TOSH 和 TOSL。在图 12-5 中 STKPTR 的上方可以看到它们。通过指令 PUSH 和 POP，程序员可以将当前程序计数器值压入栈，或者从栈顶弹出并送入程序计数器。

12.6.3 快速寄存器栈

18 系列结构不仅提供有程序计数器所需的栈，还(以原始方式)为 STATUS、WREG 和 BSR 寄存器提供了独立"栈"。它们分别占用 1 个存储器单元，合称为快速寄存器栈(Fast Register Stack)，程序员无法直接访问。当低优先级或高优先级中断发生时，上述 3 个寄存器将被保存。当选择由中断"快速"返回时，将弹出栈值。也就是说，使用 retfie 指令时需要将操作数 s 设置为 1。在 12.7.7 节中还将对这一问题进行讨论。

352

快速寄存器栈也可以用于子例程的调用与返回。如附录 5 所示，在使用 call 与 return 指令时都可以将 s 设置为 1，从而启用快速寄存器栈。当然，只有在当前未使用中断的情况下才能保证程序的安全执行。否则，子例程执行中如果发生中断，快速寄存器栈将被覆盖。

12.7 中断

18FXX2 微控制器提供了更为复杂的中断结构，相对于 16 系列有了很大的提高。它引入了第 2 个中断向量，可以划分高优先级向量和低优先级向量，这可以在图 12-6 的程序存储器映射中看到。除一个中断之外，其他所有中断都可以分配为高优先级或低优先级。此外，还提供了更多的外部中断，借助"快速返回"中断可以实现上下文自动保存。

12.7.1 中断结构概览

中断结构如图 12-7 所示。看起来相当复杂，但理解之后就可以有效地使用微控制器中断。中断源显示在图的左端。图的右端是 3 个主要的输出，其中的 2 个输出流向中断向量。激活这 2 个输出中的任何一个将导致 CPU 中断，并从图 12-6 所示的某个中断向量的位置开始执行中断服务程序(Interrupt Service Routine, ISR)。另外，还有一个"唤醒"输出，用于在休眠模式下执行。

从图中可以看到以下特性：

- ☐ 中断源(注意，并非所有中断源都已列出)；
- ☐ 中断源使能逻辑；
- ☐ 中断源优先级逻辑；
- ☐ 总体优先级使能逻辑；
- ☐ 总体(全局)使能逻辑。

12.7.2 中断源的启用与优先级划分

首先来认识部分中断源。查看标号为"外部中断源与定时器 0"的模块，它包括 5 个与门，除 1 个以外都具有 3 个输入。查看最上面的那个，它具有输入 **TMR0IF**、**TMR0IE**、**TMR0IP**。这些输入都与定时器 0 有关，这种模式在图中其他位置出现了很多次。输入…IF 是中断标志(Interrupt Flag)位，当定时器 0 发生中断时被置位；输入…IE是中断使能(Interrupt Enable)位，对应于某个 SFR 中的特定位，可以被程序置位或清零。输入…IP 是中断优先级位，也位于 SFR 中，可以被程序置位或清零。如果所有这些输入都为高电平，也即中断启用、被选为高优先级并且中断标志被置位，则与门将输出高电平，该中断将被送入下一级门电路。注意，模块中所有与门的输出通过或门连接在一起。

353

上述模块在图的下部又重复了一次，但是定时器 0 所对应的与门的第 3 个输入变成了$\overline{\text{TMR0IP}}$。可以看到，每个中断源(有 1 个例外，下面将提到)都在图的上半部和下半部各出现了一次。在上半部分中，当优先级位为高电平时中断启用；在下半部分，当优先级为低时中断启用。同样地，该模块中所有与门的输出都连在一个或门上。

不能设置优先级的中断源是外部中断 0。如图所示，它永远属于高优先级。

来自微控制器外围设备的中断位于图的左侧。它们的中断逻辑与外部中断类似，同样可以设置为高优先级或低优先级。由于这类中断比较多，因此并没有全部列出。图的左上部分画出了高优先级中断源的一般模式，低优先级中断源与此类似，显示在图的左下部。与前类似，上述逻辑确保了每个中断源都可以选择作为高优先级中断或低优先级中断。图中给出了定时器 1 的输入作为例子。每个方框中所有与门的输出同样通过或门连接在一起。

12.7.3　总体中断优先级启用

虽然上述逻辑为每个中断源提供了选择优先级的可能，但是，有时可能不需要使用这项功能。因此，需要启用或禁止整个优先级分配过程。此时，可以使用 IPEN 来完成，也即中断优先级使能(Interrupt Priority Enable)位。IPEN 是 RCON 寄存器的 MSB (最高位)，如图 12-14 所示。

现在，注意图 12-7 中间标为"优先级控制逻辑"的方框。方框中 IPEN 出现了 3 次(其中 1 个被错标为 IPE)。如果 IPEN 为低电平，来自图中下半部分的中断(包括来自外围设备或外部的中断源中断)都将上行至图的上半部分，并穿过方框中的 2 个与门。在这种情况下，所有中断都将传送至高优先级向量，因而事实上并不区分优先级。在这种状态下，还有一级启用，下面很快就会讲到。在上述 IPEN 为低电平的情况下，中断系统与 16 系列兼容。

如果 IPEN 为高电平，则高优先级中断与低优先级中断将分别到达各自所对应的向量。中断优先级启用时，各个中断源都可以通过设置各自的优先级控制位而置于低优先级或高优先级域中。

12.7.4　全局启用

全局中断启用可分为 2 个级别，由 GIE/GIEH 和 PEIE/GIEL 所控制。正如它们的名字所体现的那样，它们对于不同的 IPEN 状态表现出 2 种不同的功能。

354

如图所示，GIE/GIEH 同时控制 2 个连接到中断向量的与门，从而扮演"全局中断启用"的角色。如果 GIE/GIEH 为低电平，无论 IPEN 状态是什么，将不产生任何中断。当 IPEN 为低电平时，所有中断都将到达高优先级向量，此时它执行"全局启用"的功能。当 IPEN 为高电平时，它仍然可以同时禁止高优先级和低优先级中断。但它自己无法启用低优先级中断，因为这涉及 PEIE/GIEL。因此，它只能启用高优先级中断。

当 IPEN 为低电平时，PEIE/GIEL 使能线可以启用所有的未屏蔽外围设备中断。它通过"优先级控制逻辑"中心的或门来实现这项功能。由于它与低优先级向量所对应的那个输出与门相连，因此当 IPEN 为高电平时，它可以实现对低优先级输入的"全局启用"。

12.7.5　中断逻辑的其他方面

电路中还有 2 个需要注意的地方。从高优先级中断的输出有一根线下行至低优先级控制逻辑。可以看出，它的作用是在有高优先级中断被确认发生时将低优先级通路阻塞。但是，反之并不成立。这样高优先级中断就可以打断低优先级中断。另外，每条中断通路都有一条线路上行至或门，并连向"休眠中唤醒"。注意这些线路的功能与那 2 条"全局启用"线路的状态无关。

12.7.6 中断寄存器

从图 12-7 数量众多的位可以清楚地知道，要容纳它们需要大量寄存器。每个中断源(有 1 个例外)都需要有符号位、使能位和优先级位，另外还需要容纳所有控制位。除此之外，还需要对某些输入进行控制，例如设置外部中断的激活沿。

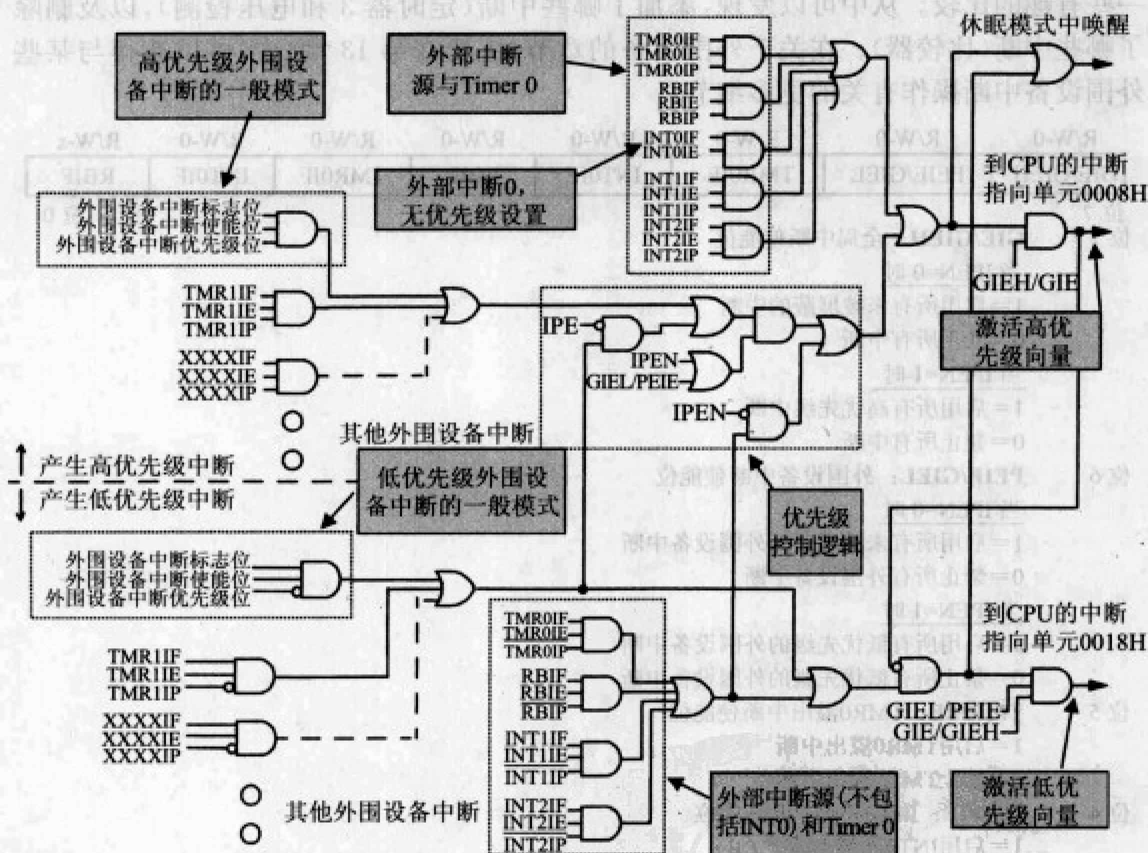


图 12-7 PIC 18F242 的中断逻辑(阴影框中所附标签为作者所加)

在中断寄存器的设计方式中，尽最大可能保留了 16 系列所用的寄存器，从而使系统设计员能够相对容易地转向 18 系列。因此，18 系列中的 INTCON 寄存器(图 12-8)几乎与 16F87XA 的寄存器 INTCON(见图 7-11)完全相同。对个别位进行了重命名，并扩展了 GIE 和 PEIE 的功能(如前所述)。

除 INTCON 寄存器之外，还添加了 2 个中断控制寄存器 INTCON2 和 INTCON3，分别显示在图 12-9 和图 12-10 中。它们用于容纳寄存器 INTCON 中出现的中断所对应的控制位。这些位的具体含义已显示在图中。另外，还有一个与众不同的位 $\overline{\text{RBPU}}$。它与中断无关，只是用来控制端口 B 的上拉，它在 16 系列中位于选项寄存器(Option

Register)，该寄存器在 18 系列中已经不存在。

外围设备中断源所对应的使能位、标志位（或请求位）和优先级位位于 PIE1 寄存器、PIE2 寄存器、PIR1 寄存器、PIR2 寄存器、IPR1 寄存器和 IPR2 寄存器中。图 12-11 和图 12-12 对此做了总结。同样地，为了实现向上兼容，它们与 16 系列中的同名寄存器非常相似（当然不包括优先级寄存器）。返回第 7 章查看图 7-12 和图 7-13，可以进行一些有趣的比较。从中可以发现，添加了哪些中断（定时器 3 和电压检测），以及删除了哪些中断（比较器）。在关于外围设备的章节（主要在第 13 章）中，可以获得与某些外围设备中断操作有关的更多细节。

355

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF
位 7							位 0

位 7 **GIE/GIEH**：全局中断使能位

当 IPEN=0 时

1=启用所有未被屏蔽的中断

0=禁止所有中断

当 IPEN=1 时

1=启用所有高优先级中断

0=禁止所有中断

位 6 **PEIE/GIEL**：外围设备中断使能位

当 IPEN=0 时

1=启用所有未被屏蔽的外围设备中断

0=禁止所有外围设备中断

当 IPEN=1 时

1=启用所有低优先级的外围设备中断

0=禁止所有低优先级的外围设备中断

位 5 **TMR0IE**：TMR0溢出中断使能位

1=启用TMR0溢出中断

0=禁止TMR0溢出中断

位 4 **INT0IE**：INT0外部中断使能位

1=启用INT0外部中断

0=禁止INT0外部中断

位 3 **RBIE**：RB端口电平变化中断使能位

1=启用RB端口电平变化中断

0=禁止RB端口电平变化中断

位 2 **TMR0IF**：TMR0溢出中断标志位

1=TMR0寄存器已经溢出（必须用软件清零）

0=TMR0寄存器没有溢出

位 1 **INT0IF**：INT0外部中断标志位

1=INT0外部中断发生（必须用软件清零）

0=INT0外部中断未发生

位 0 **RBIF**：RB端口电平变化中断标志位

1=RB7:RB4引脚中至少有1个电平状态发生改变(必须用软件清零)

0=RB7:RB4引脚状态都未发生改变

注：引脚上电平变化的情况会不断地将RBIF位置位，而读取PORTB将结束这种引脚变化的情况，并将RBIF位清零

图 12-8 PIC 18FXX2 的 INTCON 寄存器

R/W-1	R/W-1	R/W-1	R/W-1	U-0	R/W-1	U-0	R/W-1
RBPU	INTEDG0	INTEDG1	INTEDG2	—	TMR0IP	—	RBIP
位 7							位 0

位 7　**RBPU**：PORTB上拉使能位
1＝禁止所有PORTB上拉
0＝根据各个端口锁存器值启用PORTB上拉

位 6　**INTEDG0**：外部中断0触发沿选择位
1＝上升沿触发中断
0＝下降沿触发中断

位 5　**INTEDG1**：外部中断1触发沿选择位
1＝上升沿触发中断
0＝下降沿触发中断

位 4　**INTEDG2**：外部中断2触发沿选择位
1＝上升沿触发中断
0＝下降沿触发中断

位 3　未实现：读作“0”

位 2　**TMR0IP**：TMR0溢出中断优先级位
1＝高优先级
0＝低优先级

位 1　未实现：读作“0”

位 0　**RBIP**：RB端口电平变化中断优先级位
1＝高优先级
0＝低优先级

图 12-9　PIC 18FXX2 的 INTCON2 寄存器

R/W-1	R/W-1	U-0	R/W-0	R/W-0	U-0	R/W-0	R/W-0
INT2IP	INT1IP	—	INT2IE	INT1IE	—	INT2IF	INT1IF
位 7							位 0

位 7　**INT2IP**：INT2外部中断优先级位
1＝高优先级
0＝低优先级

位 6　**INT1IP**：INT1外部中断优先级位
1＝高优先级
0＝低优先级

位 5　未实现：读作“0”

位 4　**INT2IE**：INT2外部中断使能位
1＝启用INT2外部中断
0＝禁止INT2外部中断

位 3　**INT1IE**：INT1外部中断使能位
1＝启用INT1外部中断
0＝禁止INT1外部中断

位 2　未实现：读作“0”

图 12-10　PIC 18FXX2 的 INTCON3 寄存器

位 1　**INT2IF**：INT2外部中断标志位

1=INT2外部中断发生(必须用软件清零)

0=INT2外部中断未发生

位 0　**INT1IF**：INT1外部中断标志位

1=INT1外部中断发生(必须用软件清零)

0=INT1外部中断未发生

图 12-10 (续)

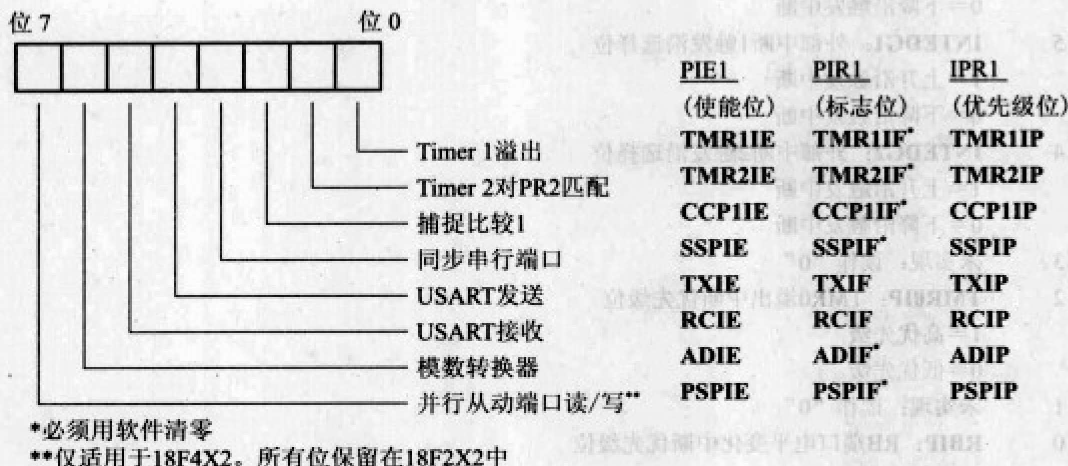


图 12-11　18FXX2 的 PIE1/PIR1/IPR1(外围设备中断启用/外围设备中断请求/外围设备中断优先级)寄存器

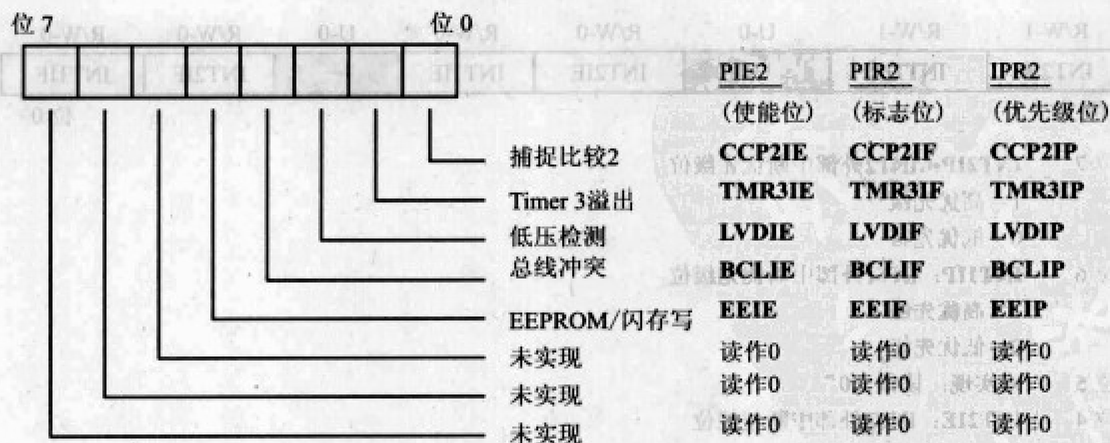


图 12-12　18FXX2 的 PIE2/PIR2/IPR2(外围设备中断启用/外围设备中断请求/外围设备中断优先级)寄存器

12.7.7 中断的上下文保护

本章 12.6.3 节所述的快速寄存器栈,在某些情况下可以轻松简便地完成上下文保护。程序员需要首先确定保存在这个栈中的 3 个寄存器 WREG、STATUS 和 BSR 是否够用。如果不够用,或者未使用从中断快速返回的功能,那么程序员就需要编写相应代码,从而在 ISR 起始处保存所有必要的寄存器,并在结尾处重新取出它们。另外,特别需要注意的是,高优先级中断可以中断低优先级中断。当这种情况发生时,高优先级中断将覆盖快速寄存器栈的内容,导致低优先级中断丢失上下文。此时,对低优先级中断使用快速寄存器栈并不安全。这类中断的上下文应当保存在软件中。

12.8 电源与复位

12.8.1 电源

图 12-13 列出了 18LFXX2 和 18FXX2 所需的电源电压。从中可以看出,18LFXX2 器件的工作电压为 2.0V～5.5V,而 18FXX2 的工作电压为 4.2V～5.5V。但是,低功耗器件无法在低电压下全速运行。参考文献 12.1 中的数据表明,它在最低电源电压下的最大时钟频率为 4 MHz。在 4.2V 时可达到 40 MHz。

12.8.2 上电与复位

2.8 节已经分析了简单 PIC 控制器 16F84A 的复位电路。18FXX2 控制器的复位结构直接由图 2-11 所示模型构建而来,只是添加了一些复位源,包括栈上溢或下溢复位、欠压复位(在 16F873A 中已经出现过)和软件复位。

356～358

除了添加更多的复位源之外,18 系列还有一些有趣的特性,它可以提供一些有用的信息用于判断复位源。因此,在复位状态之后,它并不像简单微控制器那样执行全新的启动。此时,可以查找被强制复位的原因。这对于某些情况是非常有价值的,例如在看门狗定时器超时溢出时。这种信息通过 RCON 寄存器来提供,借助其中的某些位可以确定最近发生的复位类型。RCON 寄存器对我们来说并不陌生,它的最高位是 IPEN。

图 12-14 给出了 18FXX2 的复位清单。同时,图中还列出了复位发生后的程序计数器值、RCON 寄存器位和 2 个栈溢出位(如本章 12.5 节所述)。这样,在程序重新启动时,就可以通过测试 RCON 的状态对特定复位类型作出相应的反应。

执行复位指令(表 A5-1)可以引发图 12-14 所示的软件复位。这与在输入脚$\overline{MCLR}$上施加逻辑 0 产生的外部复位相同。确保上电复位所需的条件显示在图 12-13 中,另外图中还列出了欠压复位各种可能的设置。

PIC18LFXX2（工业级）			标准运行条件（除特别声明之外）运行温度 $-40℃ \le T_A \le +85℃$（工业级）				
PIC18FXX2（工业级，扩展级）			**标准运行条件（除特别声明之外）运行温度 $-40℃ \le T_A \le +85℃$（工业级） $-40℃ \le T_A \le +125℃$（扩展级）**				
参数编号	符号	特性	最小值	典型值	最大值	单位	条件
D001	V_{DD}	电源电压					
		PIC18LFXX2	2.0	—	5.5	V	HS、XT、RC和LP振荡器模式
D001		PIC18LFXX2	4.2	—	5.5	V	
D002	V_{DR}	RAM数据保持电压①	1.5	—	—	V	
D003	V_{POR}	V_{DD}启动电压以确保内部上电复位信号	—	—	0.7	V	细节参见3.1节（上电复位）
D004	S_{VDD}	V_{DD}上升速率以确保内部上电复位信号	0.05	—	—	V/ms	细节参见3.1节（上电复位）
D005	V_{BOR}	欠压复位电压					
		PIC18LFXX2					
		BORV1:BORV0=11	1.98	—	2.14	V	85℃≥T≥25℃
		BORV1:BORV0=10	2.67	—	2.89	V	
		BORV1:BORV0=01	4.16	—	4.5	V	
		BORV1:BORV0=00	4.45	—	4.83	V	
D005		PIC18FXX2					
		BORV1:BORV0=1x	N.A.	—	N.A.	V	在器件运行电压范围之外
		BORV1:BORV0=01	4.16	—	4.5	V	
		BORV1:BORV0=00	4.45	—	4.83	V	

表中阴影旨在增强其可读性

①要在休眠模式或器件复位过程中不丢失RAM数据，V_{DD}电压应不低于此下限

图 12-13 PIC 18FXX2 的电源参数

			分别对应RCON寄存器的4、3、2、1、0位					分别对应STKPTR寄存器的7、6位	
条 件	程序计数器	RCON寄存器	$\overline{RI}$	$\overline{TO}$	$\overline{PD}$	$\overline{POR}$	$\overline{BOR}$	STKFUL	STKUNF
上电复位	0000h	0--1 1100	1	1	1	0	0	u	u
正常工作状态下的$\overline{MCLR}$复位	0000h	0--u uuuu	u	u	u	u	u	u	u
正常工作状态下的软件复位	0000h	0--0 uuuu	0	u	u	u	u	u	u
正常工作状态下的栈满复位	0000h	0--u uu11	u	u	u	u	u	u	1
正常工作状态下的栈下溢复位	0000h	0--u uu11	u	u	u	u	u	1	u
休眠状态下的$\overline{MCLR}$复位	0000h	0--u 10uu	u	1	0	u	u	u	u
WDT复位	0000h	0--u 01uu	1	0	1	u	u	u	u
WDT唤醒	PC+2	u--u 00uu	u	0	0	u	u	u	u
欠压复位	0000h	0--1 11u0	1	1	1	1	0	u	u
休眠时的中断唤醒	PC+2[1]	u--u 00uu	u	1	0	u	u	u	u

u=不变，x=未知，－=未实现位，读作“0”

图 12-14 PIC 18FXX2 的复位源以及复位发生后的程序计数器值与标志位值（阴影框中所附标签为作者所加）

12.9 振荡源

359～360

到目前为止可以发现，凡是涉及18系列硬件特性与16系列进行比较的地方，主题始终是：18系列由16系列改进而来，但具有更优的性能与更强的灵活性。同样地，18FXX2的时钟源也经历了类似的演变。

与16F87XA类似，18FXX2器件具有内部振荡器驱动电路，附带2个外部引脚OSC1和OSC2，用于连接必要的外部元件。这2个引脚可以在图12-1中看到，有趣的是，它们具有相同的附加功能。另外，外围设备Timer 1也具有振荡源，外部连接引脚标为T1OS0和T1OS1(标在旁边的附加功能也相同)。

18FXX2可以提供16F87XA所具有的全部4种时钟振荡器模式，另外还增加了4种振荡器模式，全部列在表12-6中。在配置字1中可以找到相关的配置位(见表12-4)。

下面将对振荡器模式的特性进行描述。

表12-6 振荡器模式

模式	描述	配置位 FOSC2：FOSC0
LP	低功耗	000
XT	晶体/谐振器	001
HS	高速晶体/谐振器	010
RC	外部电阻/电容	011
EC	外部时钟	100
ECIO	外部时钟，OSC2配置为RA6	101
HS + PLL	使用锁相环的高速晶体/谐振器	110
RCIO	外部电阻/电容，OSC2配置为RA6	111

12.9.1 LP、XT、HS和RC振荡器模式

361～362

这几种运行模式与16系列所使用的同名振荡器模式相同，在3.5.3节已有描述。可以重新阅读这节内容作为回顾。

12.9.2 EC、ECIO和RCIO振荡器模式

在振荡器运行的EC(外部时钟)模式中，需要将一个外部时钟源与OSC1引脚相连。这对于PIC微控制器来说并不新鲜，这种运行模式在16系列中同样可行。不同的是，这里所说的EC模式关闭了内部驱动电路中的反馈器件，从而节省了一些电流。

运行模式EC仅使用一个引脚作为输入，OSC2引脚未用。而ECIO模式利用了该引脚作为端口A的一个附加位RA6。RCIO模式采用了同样的方式，RC时钟振荡器也只使用了一个引脚，留出的OSC2可用作端口引脚。

12.9.3 HS+PLL 振荡器模式

锁相环(phase-locked loop,PLL)是一种设计巧妙的模拟数字电路,主要用于将信号频率放大整数倍。PLL 已经日益频繁地用于微控制器中对时钟信号频率的处理。采用这种技术,可以让微控制器的某些部分获得比其他部分更高的运行频率,或者将微控制器运行于高于振荡器自身频率的时钟频率上。将微控制器设置为 HS+PLL 模式,可以启用 18FXX2 PLL,将振荡器信号频率放大为原来的 4 倍。因此,如果振荡器可以运行于 10 MHz,那么使用 PLL 就可以使内部时钟频率达到 40 MHz。这种方式具有降低外部电磁干扰的效果。

与晶体振荡器类似,由于 PLL 需要有限的时间以达到稳定运行状态,因此需要使用定时器来延缓 CPU 的启动,直至 PLL 进入稳态。这实际构成了 16 系列上电定时器(Power-up Timer,PWRT,如图 2-11 所示)的一部分。采用符号 T_{PLL} 来代表这段附加时间,并将其设置为 2ms。

12.9.4 时钟源切换

如果将微控制器应用于对功耗比较敏感的项目,尤其在需要连续运行较长时间的情况下,那么切换时钟频率的功能将带来很多好处。这样,在任务强度较大时可以运行于较高的频率,反之则以较低的频率运行。从某种程度上来说,这也可以降低使用休眠模式时时钟频率的急剧变化。

使用配置位 $\overline{\text{OSCSEN}}$ 可以控制是否启用时钟源切换(见表 12-4 和表 12-5)。如果启用,就可以选择主时钟振荡器(该振荡器在所有模式下都已配置)或 Timer 1 振荡器作为时钟源。通过 OSCCON 寄存器的最末位(也是唯一的位)SCS 来选择时钟源。该位为 0 时选择主振荡器,为 1 时选择 Timer 1 振荡器。

在两种不同步的不同频率振荡器之间进行切换时,应当避免在时钟信号上发生有害的短时脉冲,其重要性是不言而喻的。因此,18FXX2 内部包含专门的电路以确保切换过程不发生错误。当 SCS 位改变状态时,在下一个指令周期开始时将暂停程序执行。执行切换前将对新的振荡器信号进行计数,8 个这样的时钟周期之后才执行切换,之后 CPU 将继续执行。切换的准确细节还与所用振荡器的类型有关,在参考文献 12.1 中可以查看完整数据。

363

12.10 18F242 编程入门

在本书中,第四部分的大多数程序都是用 C 语言来完成的,因此有必要尝试仿真一些汇编程序,从而对指令集有一些初步的认识。如果您是从本书开头读起的,那么对于 MPLAB® 开发环境和仿真器 MPSIM™ 也许并不陌生。下面将使用 MPLAB 来开发一些简单的程序。如有必要,可以回顾 4.6 节。

12.10.1　使用 18 系列 MPLAB IDE

打开 MPLAB,然后使用 Configure > Select Device 选择 18F242。可以看到,所有熟悉的开发工具仍然可用。然后使用 Configure > Configuration Bits,可以看到可用的配置位个数已大大增加。在表 12-4 和表 12-5 中可以查看这些配置位,它们在 MPLAB 中的形式如图 12-15 所示。

Configuration Bits

Address	Value	Category	Setting
300001	27	Oscillator	RC-OSC2 as RA6
		Osc. Switch Enable	Disabled
300002	0F	Power Up Timer	Disabled
		Brown Out Detect	Enabled
		Brown Out Voltage	2.5V
300003	0F	Watchdog Timer	Enabled
		Watchdog Postscaler	1:128
300005	01	CCP2 Mux	RC1
300006	85	Stack Overflow Reset	Enabled
		Low Voltage Program	Enabled
300008	0F	Code Protect 00200-01FFF	Disabled
		Code Protect 02000-03FFF	Disabled
300009	C0	Data EE Read Protect	Disabled
		Code Protect Boot	Disabled
30000A	0F	Table Write Protect 00200-01FFF	Disabled
		Table Write Protect 02000-03FFF	Disabled
30000B	E0	Data EE Write Protect	Disabled
		Table Write Protect Boot	Disabled
		Config. Write Protect	Disabled
30000C	0F	Table Read Protect 00200-01FFF	Disabled
		Table Read Protect 02000-03FFF	Disabled
30000D	40	Table Read Protect Boot	Disabled

图 12-15　在 MPLAB 中设置 18F242 的配置位(图中所示为默认值)

364

12.10.2　斐波那契程序

在例程 5-6 中,使用了 16 系列 CPU 来计算斐波那契序列。

编程练习 12-1

首先在 MPLAB 中创建项目 Fibonacci-18。将本书附属资源中的例程 5-6 的源程序复制到该项目中。使用 Configure > Select Device 选择 18F242 微控制器。需要进行一些必要的修改从而使程序正确执行。18 系列指令集是 16 系列的超集,因此需要

修改的地方非常少。注意，在这个过程中，事实上调用了汇编器的默认值。例如，类似这样的直接代码替代并没有指定指令操作数“a”的值(附录5)，因此将使用默认值1。

编程练习12-1表明，仅使用16系列指令的程序就可以在18系列器件上执行。但是，这样不能发挥18系列CPU强大的新特性。例程12-1针对18系列对斐波那契程序进行了适当的修改。代码中用粗体标出了这些修改，表明使用了一些新指令。

例程12-1 斐波那契序列产生程序(针对18F242进行了修改)

```
;************************************************************************
;Fibo_18
;In a Fibonacci series each number is the sum of the two
;previous ones, e.g. 0,1,1,2,3,5,8,13,21....
;This program calculates Fibonacci numbers within an 8-bit range,
;first going up and then down.
;Program intended for simulation only, hence no input/output
;The program demonstrates addition, subtraction, compare, and conditional
;branching.
;TJW 10.10.05                                          Tested 10.10.05
;************************************************************************
;Configuration bits need not be set

       list p=18F242
       #include P18F242.inc
;no i/o ports used

;these memory locations hold the Fibonnaci series.
fib0   equu  10  ;lowest number (oldest when going up, newest when reversing
                                              ;down)
fib1    equ  11 ;middle number
fib2    equ  12 ;highest number
fibtemp equ  13 ;temporary location for newest number
counter equ  14 ;indicates which value we have reached, opening value is 3

      org 00

;Initialse BSR
       movlb 00 ;clear BSR
;preload initial values
       movlw 0
       movwf fib0
       movlw 1
       movwf fib1
       movwf fib2
       movlw 3
       movwf counter ;have preloaded the first three numbers, so start at 3
;
forward movf  fib1,0
       addwf fib2,0
       bc    reverse ;reverse down the series if we have overflowed
       movwf fibtemp ;latest number now placed in fibtemp
       incf counter,1
;now shuffle numbers held, discarding the oldest
```

```
        movff fib1,fib0
        movff fib2,fib1
        movff fibtemp,fib2
        goto forward
;when reversing down, we will subtract fib0 from fib1 to form new fib0
reverse movf  fib0,0
        subwf fib1,0
        movwf fibtemp        ;latest number now placed in fibtemp
        decf  counter,1
;now shuffle numbers held, discarding the oldest
        movff fib1,fib2
        movff fib0,fib1
        movff fibtemp,fib0
;test if counter has reached 3, in which case return to forward
        movlw 3
        cpfseq counter
        goto reverse
        goto  forward
        end
```

编程练习 12-2

从本书附属资源中找到例程 12-1 的源文件,相应地创建一个项目并进行编译。对程序进行仿真,使用 View > Special Function Registers 和 View > File Registers 来查看这两个存储区域。注意观察它们的结构与 16 系列的不同之处。滚动特殊功能寄存器窗口查看 PCL。单步执行程序,查看在执行每条指令时,PCL 是如何递增 2 的(如 12.5.2 节所述)。同时还可以观察数据寄存器中出现的斐波那契数。

366

小结

- 18 系列微控制器代表了在 PIC 设计方法上的一个极为清晰的迈进。CPU 和存储器结构从根本上进行了重新设计,同时又保留了很多外设。
- 指令集扩展为 75 条不同的指令。有力地增强了在算术、程序分支、表存取和存储器使用方面的能力。
- 在数据存储器结构方面,提供了更多的 RAM 空间,并将特殊功能寄存器集中在一起。
- 程序存储器容量得到了很大的提高,拥有更宽的地址总线,16 位指令现在被拆分为 2 个字节来存储。栈更深也更加灵活。

参考文献

12.1. PIC 18FXX2 Data Sheet. (2002). Microchip Technology Inc., DS39564B: www.microchip.com
12.2. Migrating Designs from PIC16C74A/74B to PIC18C442 (1999). Microchip Technology Inc., AN716, DS00716A.

367

第 13 章 PIC® 18FXX2 外围设备

本章的目的在于对 PIC 18FXX2 的外围设备进行介绍。对于一种包含众多外围设备的微控制器型号来说，本章的篇幅却意外地较短。这主要有 2 个方面的原因。其一，在第 12 章中已经声明，18 系列中的很多外围设备与 16 系列外围设备的相似或完全相同，本章只需在已有知识的基础上进行阐述，而不用再重复书中已有的内容。其二，由于后续章节将转向使用 C 语言来编程，可以从 C 编译器中获得很多支持；C 库函数将负责与外围设备之间的所有交互，因而对于常规使用，就无需了解太多与外围设备相关的细节。因此，先前在学习外围设备精确细节的过程中所耗费的精力就可以解放出来，并转移到编写工作程序的创造性工作上来。这是非常有意义的一步。

在本章的叙述中，会反复提到前几章中的 16 系列外围设备。如果你对这些外围设备已经非常熟悉，就可以以此作为有效的复习，并从中发现 18 系列所引入的变化。如果你对 16 系列外围设备并不了解，也可以以这种方式进行学习(包括 18 系列外围设备)。

18FXX2 并不能包含关于 18 系列的所有知识，因此，在本章的最后一节对该型号的另一个成员 18F2420 进行了介绍。我们需要了解它所具有的一些有趣特性，尤其是扩展指令集和纳瓦技术。

本章主要包括：

- ☐ 概述 PIC 18FXX2 的所有外围设备，多数情况下将利用 16 系列中同型外围设备的有关知识；
- ☐ 简单地了解 18 系列中一些可用的增强特性。

对本章内容可以进行通读，也可以简单地跳过，只将其作为后续章节的参考，在必要时进行查阅。

13.1 18FXX2 外围设备概述

表 12-1 对 18FXX2 微控制器的外围设备进行了总结，在图 12-2 中可以看到 2 种 18F2X2 类型器件的外围设备。当把它们与图 7-2 中的 16F87XA 外围设备进行比较时，可以看出，几乎在所有情况下都出现了相同的外围设备，且名称和模式都一样。其中需要注意的差别是，18FXX2 包含一个附加定时器 Timer 3，但却不具备 16F87XA 的比较器模块。

13.2 并行端口

18FXX2 的并行端口在结构与接口上与 16 系列非常相似。每个并行端口都有一个 **PORTX** 寄存器用于传输数据，一个 **TRISX** 寄存器用于设置数据方向。只有一个显著的差别需要注意，下面将对此做简短的说明。

在使用端口时，需要做 4 件事：设置数据方向，读取输入值，设置输出值以及读回先前写入的输出值。其中，16 系列的设计有一个缺点，它对上述的第 4 项工作并不擅长。假设某个端口位(例如图 7-15a)被设置为输出，并向其写入了位值。如果 CPU 对该位进行读取，就无法确定所读取到的值是否等于先前所写入的值。这是因为所读取的是实际端口位引脚的值。这个值可能是位电路的输出值，也可能是由引脚上所连的外部器件强制施加的值。

为了解决这个问题，18 系列端口进行了一个有趣的改进。每个端口增加了第 3 个寄存器用于保存锁存输出端口位的值，端口 A 对应 **LATA**，端口 B 对应 **LATB** 等。该寄存器可以由程序来读取，程序员完全可以确定，所读出的值就是该端口先前所保存的值。

下面将逐个介绍 18FXX2 的端口，并讲解其中涉及的各种更改。

13.2.1 18FXX2 的端口 A

在图 12-1 所示的引脚排列图中可以看到端口 A 的引脚位置。从中可以清楚地看到，端口模式与 16F873A 几乎相同，除用作数据 I/O 端口之外，还可复用为 ADC(模数转换器)输入。端口的第 4 位可复用为 Timer 0 的输入，为漏极开路(Open Drain)输出方式。如果使用了 ECIO 或 RCIO 振荡器模式，还可以使用一个额外的端口位 RA6，如 12.9.2 节中所述。

在寄存器映射(见图 12-5)中的第 4 列，可以看到与端口 A 有关的 3 个寄存器 **PORTA**、**TRISA** 和 **LATA**。图 13-1 描述了新加的数据锁存寄存器 LATA 的工作原理。这张图与 16 系列中的对应图(图 7-15a)非常相似。新加入 LATA 寄存器之后，读者可能会希望在电路中某个地方找到双稳态触发器。但事实上它并不存在。大致浏览电路图会发现，**LATA** 和 **PORTA** 共同使用同一个锁存器。向其中一个写入数据等价于向另一个也写入数据。电路中的唯一差别事实上在于图顶端所出现的 **LATA** 缓冲器。对 **LATA** 的读操作将激活该缓冲器，并将 **PORTA/LATA** 数据锁存器中保存的数据传送至数据总线。因此，**LATA** 并不是一个新的寄存器，只是在寻址时可以直接读取 **PORTA** 数据锁存器的输出。

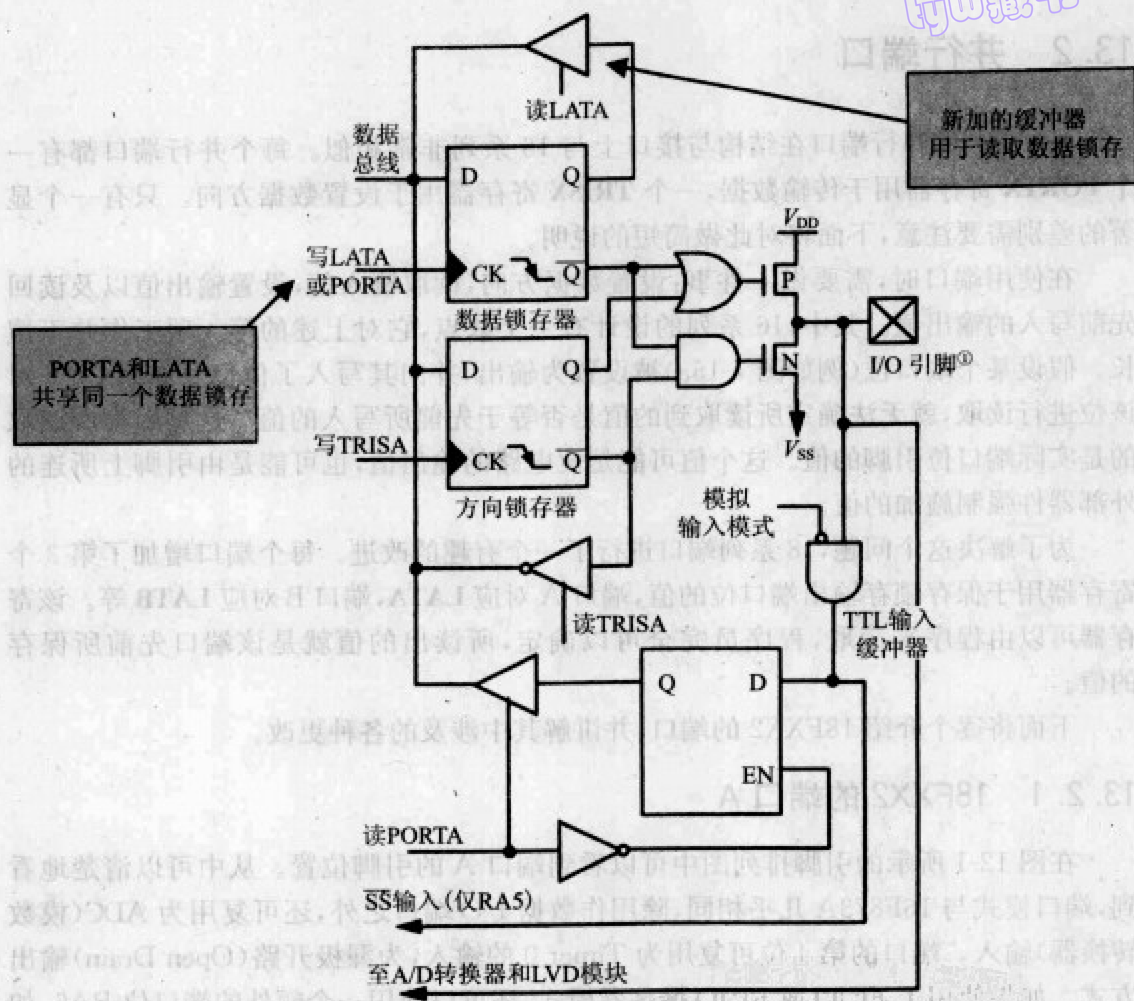


图 13-1　18FXX2 RA0～RA3 和 RA5 引脚(阴影框中所附标签为作者所加)

13.2.2　18FXX2 的端口 B

在图 12-1 所示的引脚排列图中可以看到端口 B 的引脚位置，它的 3 个主要寄存器 **PORTB、TRISB** 和 **LATB** 显示在图 12-5 中。**LATB** 的功能与上述 **LATA** 的功能相同。

369

该端口保留了 16 系列的大多数特性，在设计上更为直接。添加了更多的可复用功能，特别是引入了更多的外部中断源。如图所示，位 5～7 可分别复用为电路内调试功能 **PGD、PGC** 和 **PGM**。与 16 系列相比，PGM 的位置已经发生了改变。端口的低 3 位可分别复用为 3 种外部中断输入。此外，在位 3 上还额外引入了一个可选的 CCP2 连接。这样就可以缓解端口 C 中位 1 引脚过于“紧张”的压力，从而在使用 CCP2 的同

时使用 Timer 1 外部振荡器。

通过控制位$\overline{\text{RBPU}}$可以在所有引脚上实现内部上拉，此控制位位于寄存器 **INTCON2** 中(如图 12-9 所示)。另外端口 B 的某些引脚还具有电平变化中断功能，也即引脚 4 到引脚 7 中任一引脚上的电平发生变化，都将导致相应中断符号位被置位。该中断使能位和标志位 **RBIE** 和 **RBIF** 位于寄存器 **INTCON** 中(见图 12-8)。 370

13.2.3 18FXX2 的端口 C

与其他端口类似，在图 12-1 所示的引脚排列图中可以看到端口 C 的引脚位置，它的 3 个主要寄存器 **PORTC**、**TRISC** 和 **LATC** 显示在图 12-5 中。与 16F87XA 的情况相同，此端口引脚被复用为串行端口和 CCP 功能，位 0 则复用为 Timer 1 的输入。位 0和位 1 也可用作 Timer 1 的外部振荡器输入。由于复用功能较多，端口引脚驱动电路也相当复杂。此外，引脚功能还可以被外围设备功能所替代，因此必须小心。但是，这种替代不包括对**LATC**寄存器的读取。无论端口引脚运行模式是什么，都可以读取**LATC**寄存器。

13.2.4 并行从动端口

这种端口出现在有 40 个引脚的器件中，与 7.12 节所述特性相同。并行端口自身为端口 D，另外使用端口 E 中的 3 位作为握手信号。端口的运行模式主要由寄存器 **TRISE** 控制，与图 7-26 相同。端口中仅有的更改是新增了 **LATD** 寄存器，其功能与上述端口 A 所述内容相同。

13.3 定时器

18FXX2 中所有类型的器件都具有 4 个可编程定时器和 1 个看门狗定时器。下面将逐个对这些定时器进行讲解，其中会涉及 16 系列同名定时器的很多内容。

13.3.1 Timer 0

18FXX2 的 Timer 0 与 16 系列的 Timer 0 有很大的区别。它可以运行于 8 位或 16 位模式。图 13-2 列出了它所对应的控制寄存器 **T0CON**。其中的低 6 位替代了 16 系列 **OPTION** 寄存器(见图 6-9)，二者功能类似。通过位 6 可以选择总的运行模式，也即运行于 8 位模式还是 16 位模式。

在 8 位模式下，Timer 0 的动作与图 6-8 所示 16 系列 Timer 0 的动作相同。但 16 系列的 Timer 0 与 WDT 共用一个预分频器，而在 18 系列中，该分频器完全属于 Timer 0。因此，**T0CON** 中的 **PSA** 位仅用于确定是否使用预分频器，而不会将其分配给看门狗定时器。

定时器在 16 位模式下的动作如图 13-3 所示。计数器自身的低字节称为 **TMR0L**, 371

高字节则直接称为 **TMR0**，因为它与 8 位模式下的寄存器地址相同。

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
TMR0ON	T08BIT	T0CS	T0SE	PSA	T0PS2	T0PS1	T0PS0
位 7							位 0

位 7　**TMR0ON**：Timer 0开/关控制位
1=启用Timer 0
0=禁用Timer 0

位 6　**T08BIT**：Timer 0 8位/16位控制位
1=Timer 0被配置为8位定时器/计数器
0=Timer 0被配置为16位定时器/计数器

位 5　**T0CS**：Timer 0时钟源选择位
1=T0CKI引脚上的输入信号作为时钟信号源
0=内部指令周期时钟(CLKO)

位 4　**T0SE**：Timer 0时钟源边沿选择位
1=T0CKI引脚信号下降沿时递增
0=T0CKI引脚信号上升沿时递增

位 3　**PSA**：Timer 0预分频器分配位
1=Timer 0不用预分频器，Timer 0时钟输入避开预分频器
0=Timer 0使用预分频器，Timer 0时钟输入来自预分频器输出

位 2~0　**T0PS2:T0PS0**：Timer 0预分频器倍率选择位
111=1:256预分频值
110=1:128预分频值
101=1:64预分频值
100=1:32预分频值
011=1:16预分频值
010=1:8预分频值
001=1:4预分频值
000=1:2预分频值

图 13-2　Timer 0 的控制寄存器 T0CON

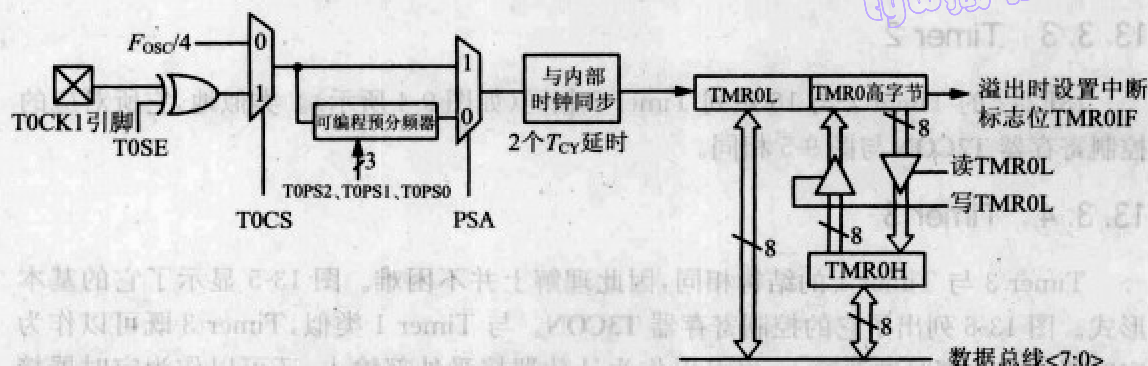
16 位定时器运行于 8 位环境下会出现一个问题。假设程序需要读取定时器中保存的值，将会逐个读取 2 个字节。但是，有时会发生下述情况：首个字节读出后计数器值又增加 1，有可能会导致低字节向高字节的进位溢出。这样，读出的 2 字节值就会产生严重错误。

图 13-3 给出了这个问题的解决方法。在定时器高字节 **TMR0** 旁边又添加了一个缓冲器 **TMR0H**。高字节是无法直接读取的，但是每当读取低字节时，**TMR0** 的值就同时传送至 **TMR0H**，该缓冲器的值可以在后续指令中读取。采用这种方式，就可以确保所读出的高字节值与低字节值为同一时刻的值，即进行读操作时的值。与此类似，当程序员要写入定时器时，应当首先在程序中将所需的高字节写入 **TMR0H**。当低字节写入 **TMR0L** 时，**TMR0H** 中保存的值将同时传送至 **TMR0**。同样地，按照这种方式可以确保向定时器传送正确的取值，而不会被 2 字节传输过程中的递增动作所干扰。

372

无论处于哪种模式下，Timer 0 都将在计数器从最大值（8 位时为 FF_H，16 位时为 $FFFF_H$）溢出时产生中断。这两种中断使用相同的标志位**TMR0IF**（如图 12-7 所示）。

在后续几章的某些 C 语言例程中，将用到 16 位模式下的 Timer 0 及其中断。

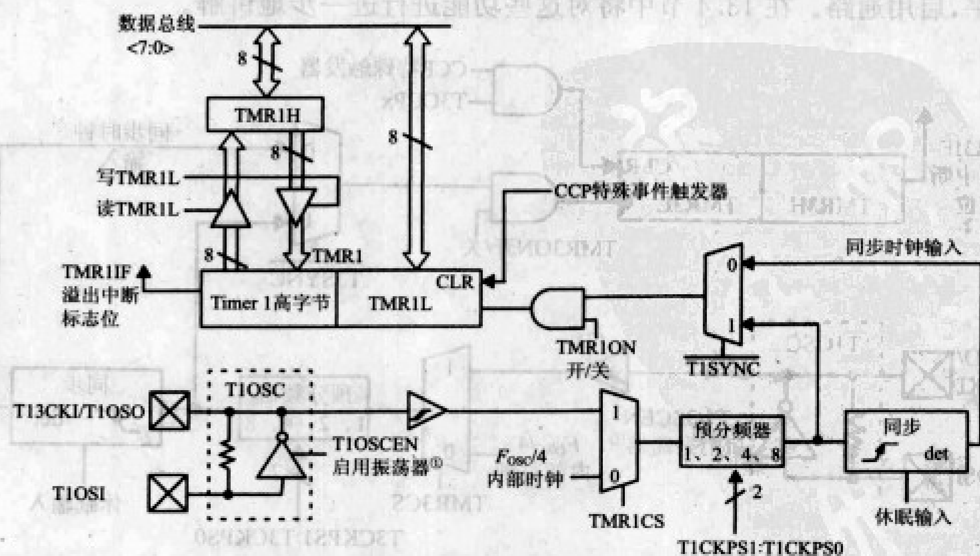


注：复位时，将启用8位模式下的Timer 0，并选择T0CKI作为时钟输入，使用最大预分频值

图 13-3 Timer 0 运行于 16 位模式

13.3.2 Timer 1

18FXX2 的 Timer 1 的基本形式与 16 系列的 Timer 1(参见图 9-1)几乎相同。它所对应的控制寄存器为**T1CON**，与图 9-2 类似，只是位 7(16 系列中未用)被称为**RD16**。当该位被置为 1 时，将启用"16 位读/写"模式。这与 Timer 0 的情况相同，该模式启用时的运行原理如图 13-4 所示。此处，为高字节配备了缓冲寄存器**TMR1H**，用于实现定时器的同步读写数据传输。另外，它与 16 系列 Timer 1 不同的地方是，它可以通过来自 CCP 模块的"特殊事件"来清零定时器。在图 13-4 中可以观察到这种情况，详细内容将在 13.4 节中讲述。



①启用位T1OSCEN清零后，将关闭反相器和反馈电阻，以降低功耗

图 13-4 运行在 16 位读/写模式下的 Timer 1

13.3.3 Timer 2

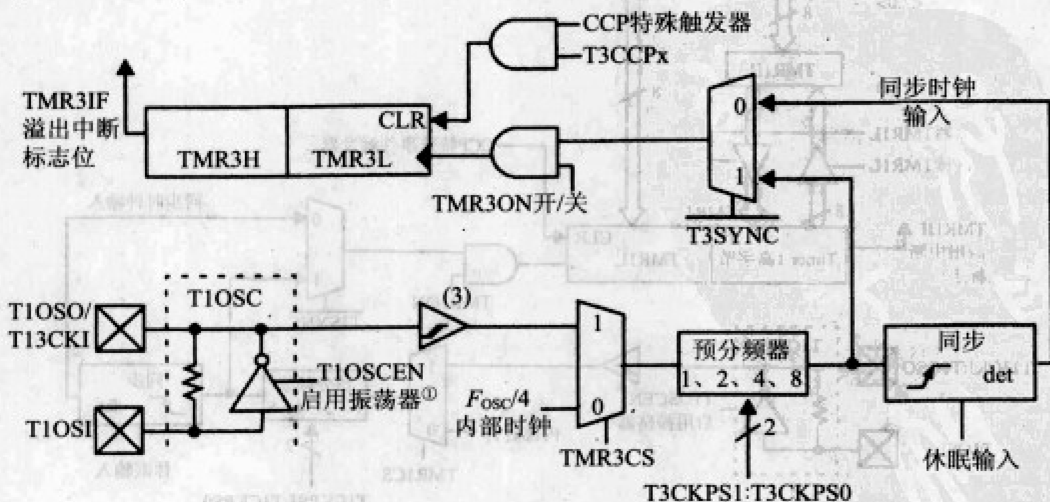
18F242 的 Timer 2 与 16 系列 Timer 2 相同(如图 9-4 所示)。类似地,它所对应的控制寄存器 **T2CON** 与图 9-5 相同。

13.3.4 Timer 3

Timer 3 与 Timer 1 的结构相同,因此理解上并不困难。图 13-5 显示了它的基本形式。图 13-6 列出了它的控制寄存器 **T3CON**。与 Timer 1 类似,Timer 3 既可以作为373定时器接受内部时钟源输入,也可以作为计数器接受外部输入,还可以作为定时器接受外部振荡器输入。对于后两种情况,它与 Timer 1 使用相同的输入。因此,当使用外部振荡器时,也即使用了 Timer 1 的外部振荡器,此时启用 **T1CON** 中的 **T1OSCEN** 位。如果使用外部输入,事实上使用了 Timer 1 的同一输入,在 18F2X2 中就是引脚 11。

仔细观察图 13-6 可以发现,位 0、位 1、位 2、位 4、位 5、位 7 的功能与 **T1CON** 的对应位相同(见图 9-2)。与 Timer 0 与 Timer 1 相同,通过位 7 同样可以设置"16 位读/写"模式。如果未选择该模式,定时器将按照图 13-5 所示运行。反之,若选择该模式,定时器与数据总线的接口将采用图 13-4 所示的"16 位读/写"形式。此时,对高字节的读写将通过缓冲器实现,该缓冲器在存储器中被映射为 **TMR3H**。

从 **T3CON** 的位 6 和位 3 可以清楚地看到,Timer 3 可以与 CCP(捕捉/比较/PWM)模块连接,用于捕捉和比较,从而替代 Timer 1,或者与 Timer 1 同时工作。图 13-5 中出现的"CCP 特殊触发器"通过这些控制位中的某一个进行门控制。如果位 3 或位 6 为高电平,启用通路。在 13.4 节中将对这些功能进行进一步地讲解。



①启用位T1OSCEN清零后,将关闭反相器和反馈电阻,以降低功耗

374

图 13-5 Timer 1 的框图

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
RD16	T3CCP2	T3CKPS1	T3CKPS0	T3CCP1	$\overline{\text{T3SYNC}}$	TMR3CS	TMR3ON
位 7							位 0

位 7　**RD16**：16位读/写模式启用位
1=启用Timer 3通过1次16位操作实现寄存器读写
0=启用Timer 3通过2次8位操作实现寄存器读写

位 6和位3　**T3CCP2:T3CCP1**：Timer 3和Timer 1到CCPx的启用位
1x=Timer 3是比较/捕捉CCP模块的时钟源
01=Timer 3是比较/捕捉CCP2模块的时钟源
Timer 1是比较/捕捉CCP1模块的时钟源
00=Timer 1是比较/捕捉CCP模块的时钟源

位 5和位4　**T3CKPS1:T3CKPS0**：Timer 3输入时钟预分频选择位
11=1:8预分频值
10=1:4预分频值
01=1:2预分频值
00=1:1预分频值

位 2　**$\overline{\text{T3SYNC}}$**：Timer 3外部时钟输入同步控制位
（不适用于系统时钟来自Timer 1/Timer 3的场合）
当TMR3CS=1时
1=不同步外部时钟输入
0=同步外部时钟输入
当TMR3CS=0时
此位被忽略。TMR3CS=0时Timer 3使用内部时钟

位 1　**TMR3CS**：Timer 3时钟源选择位
1=使用Timer 1振荡器输出或T1CKI作为外部时钟输入
（第1个下降沿后的上升沿计数）
0=内部时钟（*Fosc*/4）

位 0　**TMR3ON**：Timer 3使能位
1=启用Timer 3
0=停止Timer 3

图 13-6　Timer 3 的控制寄存器 T3CON

13.3.5　看门狗定时器

看门狗定时器(Watchdog Timer，WDT)在概念上与16系列WDT相同，见6.5节。它是一种自由运行的递减计数器，如果在启用状态下发生溢出，将导致微控制器复位。通过配置寄存器**CONFIG2H**中的**WDTEN**位(见表12-4)来启用WDT。它具有自己的专用后分频器(在16系列中，WDT与Timer 0共享同一个后分频器)，相应设置由同一寄存器中的**WDTPS2**至**WDTPS0**位所决定。参考文献12.1指出，当后分频值为1时，溢出周期的典型值为18ms，最小值为7ms，最大值为33ms。如果将后分频值设置为128，典型溢出值可以达到2.3s。执行**clrwdt**指令将同时对WDT和后分频器清零。

18系列在WDT设计方式上的重要改进就是添加了软件WDT使能位**SWDTEN**。这是**WDTCON**寄存器中的最低位，也是唯一的可用位，在图12-5中位于存储器单元$FD1_H$。如果在配置寄存器中禁止了WDT，那么可以通过设置**SWDTEN**位来启用

WDT。反之,如果在配置寄存器中启用了 WDT,SWDTEN 位就不起作用。WDT 所具有的开关能力似乎与 WDT 的整个理念是背道而驰的,很难看出关闭一种安全特性有什么明智之处。但是,这种能力使得 WDT 可以在某些运行模式下启用,而在另一些模式下禁止。与所有安全特性或可靠性特性类似,在使用 WDT 时要加倍小心。

13.4 比较/捕捉/PWM(CCP)模块

如图 12-2 所示,18FXX2 具有两个 CCP 模块。它们与 16 系列模块非常相似,运行方式与 9.4.1 节所述原理相同。CCP 模块与 Timer 1、Timer 2 和 Timer 3 协同完成捕捉、比较和 PWM 操作。与 16 系列相比,新加了 Timer 3 并在定时器内部与 CCP 模块互连,这是二者之间的重要差别,同时也是操作灵活性上的重大改进。

13.4.1 控制寄存器

如图 13-7 所示,CCP 模块主要由 **CCPxCON** 寄存器所控制。它们与图 9-7 中 16 系列 **CCPxCON** 寄存器非常相似。仔细观察 4 个低有效位会发现,二者在可用模式上存在细微差别。18FXX2 提供了一种新模式,对应于位 3～0 位设置为 0010 的状态。这些寄存器的位 4 和位 5 的名称发生了改变,但功能未变。

U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	DCxB1	DCxB0	CCPxM3	CCPxM2	CCPxM1	CCPxM0
位 7							位 0

位 7~6 未实现:读作“0”

位 5~4 **DCxB1:DCxB0**:PWM占空比位 1和位 0

捕捉模式

未使用

比较模式

未使用

PWM模式

它们是10位PWM占空比的2个LSB(位 1和位 0)。占空比的高8位(DCx9:DCx2)在CCPRxL中

位 3~0 **CCPxM3:CCPxM0**:CCPx模式选择位

0000=禁用捕捉/比较/PWM(复位CCPx模块)

0001=保留

0010=比较模式,匹配时翻转输出(CCPxIF位置位)

0011=保留

0100=捕捉模式,每个下降沿发生

0101=捕捉模式,每个上升沿发生

0110=捕捉模式,每4个上升沿发生

0111=捕捉模式,每16个上升沿发生

1000=比较模式

CCP引脚初始为低电平,比较匹配时强制CCP引脚输出高电平(CCPIF位置位)

1001=比较模式

CCP引脚初始为高电平,比较匹配时强制CCP引脚输出低电平(CCPIF位置位)

1010=比较模式

比较匹配时产生软件中断(CCPIF位置位,CCP引脚不受影响)

1011=比较模式

触发特殊事件(CCPIF位置位)

11xx=PWM模式

图 13-7 CCP1CON 和 CCP2CON 寄存器(地址为 FBDh 和 FBAh)

13.4.2 捕捉模式

375 ~ 376

图 13-8 显示了配置为捕捉模式的 CCP。它与图 9-8 直接等价，其原理与 9.4.2 节所述相同，只是这里画出了 2 个输入。结构上的主要差别在于，此时 Timer 1 和 Timer 3 都可以使用，并且针对它们所要"捕捉"的信号进行了频率和预分频器设置上的优化。通过 T3CON 寄存器中的**T3CCPx**位可以对定时器进行选择(如图 13-6 所示)。该选择一旦确定，捕捉操作就与 16 系列相同。

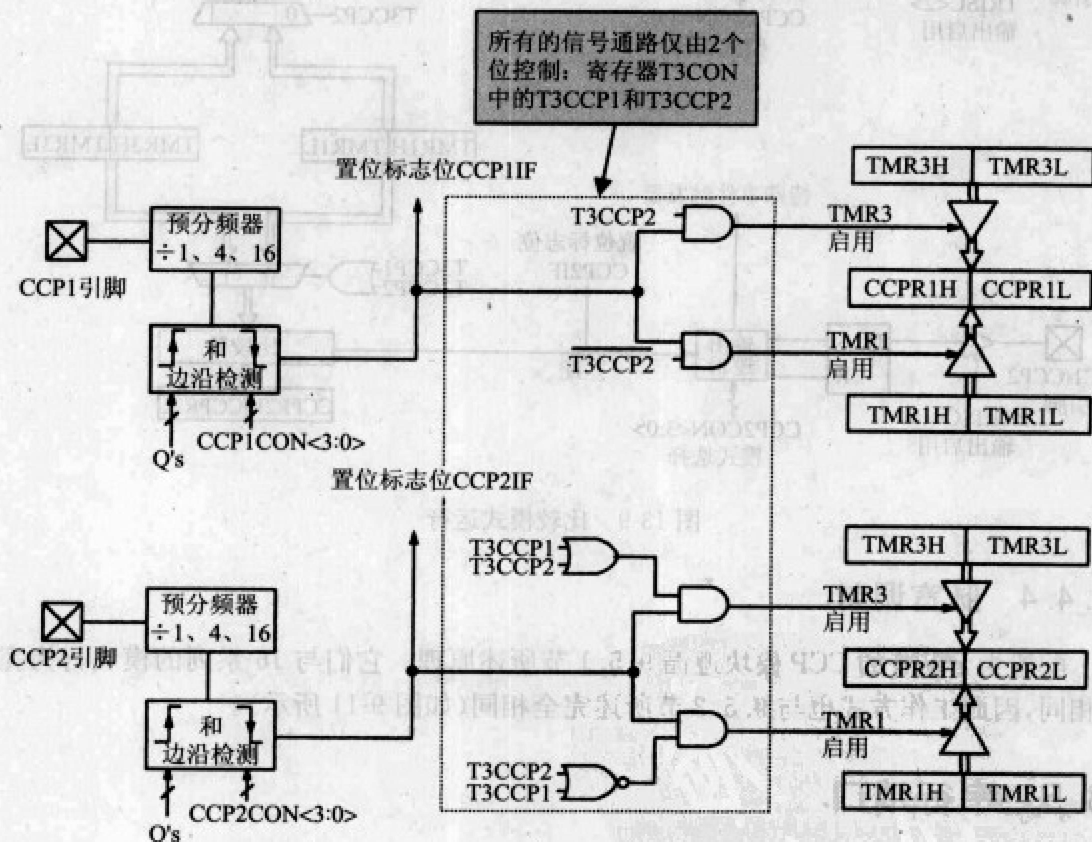


图 13-8 捕捉模式运行(阴影框中所附文字为作者所加)

13.4.3 比较模式

图 13-9 显示了配置为比较模式的 CCP。它与图 9-9 直接等价，其原理可参见 9.4.3 节。与上述捕捉模式相同，这里也画出了 2 个输入。该图还显示了由于 Timer 1 和 Timer 3 都可用而导致的结构上的主要差别。同样地，通过 **T3CCPx** 位来选择定时器(如图 13-6 所示)。

377

在图的左上角对"特殊事件"动作进行了总结。通过合理设置寄存器 **CCP1CON** 和 **CCP2CON** 中的低 4 位进行选择(见图 13-7)。

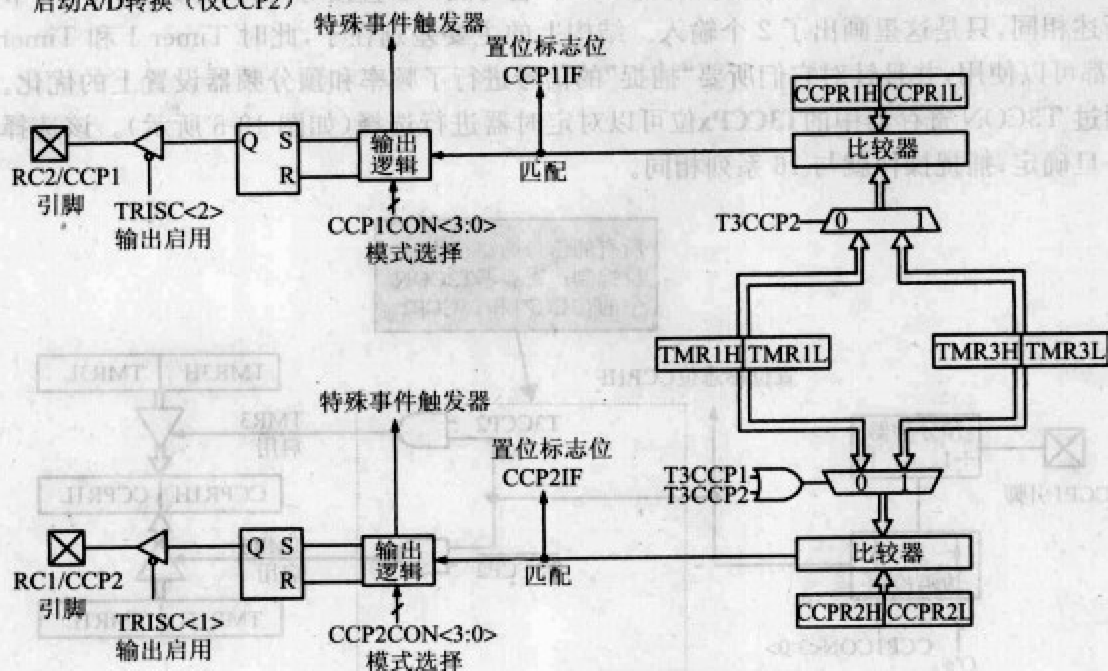


图 13-9 比较模式运行

13.4.4 脉宽调制

配置为 PWM 的 CCP 模块遵循 9.5.1 节所述原理。它们与 16 系列的模块行为完全相同，因此工作方式也与 9.5.2 节所述完全相同(如图 9-11 所示)。

13.5 串行端口

如图 12-2 所示，18FXX2 具有 2 种主要的串行端口：主同步串行端口(Master Synchronous Serial Port，MSSP)和可寻址的通用同步异步收发器(Universal Synchronous Asynchronous Receiver Transmitter，USART)。MSSP 与 16 系列中的同名外围设备相同。因此，它可以配置为 SPI(Serial Peripheral Interface，串行外围设备接口)模式或I^2C模式。下面将对两种端口的行为做简要总结。

13.5.1 SPI 模式下的 MSSP

在此模式下，MSSP 的结构如图 10-7 所示。该串行端口的数据输入与输出通过 **SSPBUF** 寄存器(如图 12-5 中的地址 FC9h)进行。该端口由 2 个寄存器 **SSPCON1** 和 **SSPSTAT** 控制，图 10-8 和图 10-9 分别进行了介绍，在 18 系列存储器映射中分别位于

378

地址 FC6h 和 FC7h(如图 12-5 所示)。10.3 节对该模式下的外围设备行为特性作了介绍。由于 18 系列引入了中断优先级(参见 12.7 节),对于 **MSSP** 中断(图 12-11 **PIR1** 中的 **SSPIF**)同样可以设置优先级。通过 **IPR1** 寄存器中的 **SSPIP** 位可以控制优先级的选择(如图 12-11 所示)。

13.5.2 I²C 模式下的 MSSP

I²C 模式下的 MSSP 与 16 系列的 MSSP 原理相同。因此,它遵循 10.6 节所述的所有 I²C 原理。它使用相同的控制寄存器 **SSPCON1**、**SSPCON2**、**SSPSTAT** 和 **SSPADD**。从动模式下的结构如图 10-17 所示,主动模式下的结构如图 10-19 所示。总体来讲,它的工作原理可参见 10.7 节。

379

13.5.3 USART

该模块与 16 系列中的同名外围设备相同,其行为特性与 10.10 节所述相同。由于具有更高的振荡器极限频率,因此可以运行在更高的波特率下。它的中断(图 12-11 **PIR1** 中的 **RCIF**、**TXIF**)可以设置优先级,遵循 18 系列的中断使能机制。

13.6 模数转换器(ADC)

18FXX2 中的 ADC 模块与 11.3 节所述的 16F873A 非常类似。其结构如图 11-6 所示,18F442 和 18F452 都有 8 个输入,18F242 和 18F252 则都有 5 个输入,在图 12-1 中可以看到它们。ADC 由寄存器 **ADCON0** 和 **ADCON1** 控制,此时映射为地址 FC2h 和 FC1h(见图 12-5)。转换结果放置在 **ADRESL** 和 **ADRESH** 中(图 12-5 中的 FC3h 和 FC4h)。图 11-10 中给出了模拟输入模型。

从图 13-9 可以看出:经设置,比较模式下的 CCP 模块可以触发 ADC 转换。为此,必须通过 **ADCON0** 中的 **ADON** 位(见图 11-7)将 ADC 模块打开。输入多路选择器必须按照 11.3.4 节所述进行预设,为数据采集留有足够的时间。如果让引发特殊事件触发的定时器(Timer 1 或 Timer 3)持续运行,就可以实现周期性的模数转换。

13.7 低压检测

在嵌入式系统中,对渐趋衰弱的电源的检测能力是很有价值的。如果产品由电池供电,就可以使用这种功能来检测电池是否即将耗尽。另外,还可以用于检测电源的突然关断。在以上这些情况下,微控制器都希望能够激活报警信号或执行有序的停机,比如将运行中的关键变量保存在 EEPROM 中。在完全掉电的情况下,系统充电电容中的残留电荷依然可以维持一段时间以完成这些操作。

在很多稳压 IC 中都具有低压检测功能,特别是在那些由电池供电的器件中。它

们将在检测到低压信号时产生输出，可以将其作为中断传给微控制器。而在 18FXX2 中，它拥有自己的低压检测功能和相应的中断源。图 13-10 显示了其一般原理。电源电压 V_{DD}施加在电阻排列的顶端。另一端可以通过开关晶体管与地连接，开关管的门电压为 **LVDEN**，位于 **LVDCON** 寄存器的位 4 中（如图 13-11 所示）。

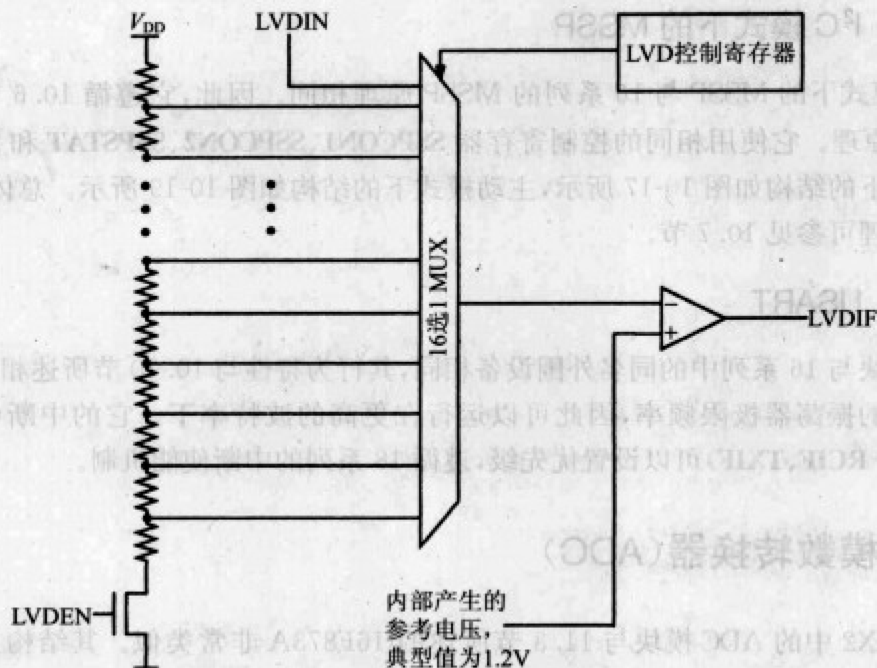


图 13-10 低压检测电路

380

当 **LVDEN** 为高电平时，电阻阵列就成为一个分压器，其中各种不同的节点与多路选择器连接。通过设置 **LVDCON** 的低 4 位可以选择其中的某个输入作为多路选择器的输出。该多路选择器输出值与内部产生的 1.2V 参考电压进行比较。如果多路选择器输出电压降到参考电压以下，那么比较器输出 **LVDIF** 将变为高电平。这是低压检测中断标志，如图 12-12 所示。如果该中断启用，此时将产生中断，微控制器就可以对电压的衰减做出适当的反应。

图 13-11 列出了所有可选的“行程（trip）”电压，可以将它们分别与图 12-13 中的器件有效运行范围和可能的欠压设置进行比较。很明显，将低压检测电压设置在欠压复位电压之下是没有用的，因为器件将在低压检测到来之前复位。因此，设计员可以选择不使用欠压复位，而对所有的电源线使用低压检测监视。

除了由内部电阻分压器来产生电压之外，还可以使用外部输入，通过低压检测输入引脚**LVDIN**（图 12-1 所有封装中的引脚 7）来引入。在图 13-10 中可以看到此信号通路，按照图 13-11 进行必要的设置。如果选择此输入，该外部电压将直接与 1.2V 的内部电压进行比较。

由于低压检测启用时将消耗一些电流（30μA 左右），因此可以通过周期地启用来

降低能耗。这种情况下,参考电压需要一定的建立时间。参考电压状态由LVDCON寄存器中的IRVST位指出。只有在该位置位时,才能启用中断。

U-0	U-0	R-0	R/W-0	R/W-0	R/W-1	R/W-0	R/W-1
—	—	IRVST	LVDEN	LVDL3	LVDL2	LVDL1	LVDL0
位 7							位 0

位 7~6　未实现：读作“0”
位 5　**IRVST**：内部参考电压稳定标志位
1=表明低压检测逻辑将会在特定电压范围产生中断标志
0=表明低压检测逻辑将不会在特定电压范围产生中断标志，不要启用LVD中断
位 4　**LVDEN**：低压检测电源启用位
1=启用LVD，给LVD电路上电
0=禁用LVD，LVD电路未上电
位 3~0　**LVDL3**:**LVDL0**：低压检测限制位
1111=使用外部模拟输入(输入来自于LVDIN引脚)
1110=4.5V~4.77V
1101=4.2V~ 4.45V
1100=4.0V ~ 4.24V
1011=3.8V ~ 4.03V
1010=3.6V ~3.82V
1001=3.5V~ 3.71V
1000=3.3V~3.50V
0111=3.0V ~3.18V
0110=2.8V ~ 2.97V
0101=2.7V ~2.86V
0100=2.5V~2.65V
0011=2.4V~2.54V
0010=2.2V~2.33V
0001=2.0V~2.12V
0000=保留

注：如果所选LVDL3:LVDL0模式导致行程点电压低于器件有效工作电压，将无法检测

图 13-11　LVDCON 寄存器

应注意的是:任何复位操作都将清除LVDEN位,从而禁止低压检测功能。

13.8　在 Derbot-18 中应用 18 系列

由于 16F873A 与 18F242 的引脚兼容,因此 Derbot 无需任何改变就能兼容 18 系列器件,包括所有并行 I/O、ADC、PWM、定时器和计数器以及 I²C。装载了 18F242 微控制器的 Derbot 适用于后续各章中的所有例程。将这类 Derbot 称为“Derbot-18”,表明使用了不同的微控制器(但并没有其他改变)。

从硬件角度来讲,使用 18F242 所能完成的任务几乎都可以使用 16F873A 来完成。前者技术上的进步体现在C语言的使用,C语言在 18 系列中能够得到更好的发挥。这也导致了实时操作系统的使用,在本书最后的几章中将会对此进行讲解。

13.9　18F2420 与扩展指令集

18FXX2 微控制器是一种设计成功且功能强大的型号,然而它们只能代表 18 系列

微控制器成就的一个方面。另外,还有一种 18 系列的型号与 18FXX2 型号非常接近。它们由一组器件构成,编号时在标识符中额外添加了 0。因此,18F242 就对应于 18F2420,18F442 对应于 18F4420 等等。每个对应组中的两个器件均拥有相同的存储器容量、类似的外围设备集合以及相同的基本引脚排列。

下面将简要介绍这类子型号的一些重要改进,因为它们所表现出的一些其他特性与 18 系列相同。在参考文献 13.1 中可以查看详细内容。

13.9.1 纳瓦技术

很多嵌入式系统由电池供电,因此尽可能地降低能耗就非常重要。本书并没有对如何减少能耗进行详细的讨论,这些内容可以在参考文献 1.1 中找到。按照文献中的解释,对于给定的电路和技术,降低能耗有 3 种简单的方法:

- ☐ 降低电源电压;
- ☐ 降低时钟频率;
- ☐ 关断未使用的电路模块。

在本书迄今所描述的所有微控制器中,都可以使用休眠模式,这是在器件未活动时降低能耗的有效方法。但是,很多情况下微控制器都需要在保持运行的同时将能耗降至最低。使用 Microchip 公司的纳瓦技术可以达到这一要求。下面简要列出了它所具有的特性。

1. 可选的运行模式

微控制器不需要始终运行在主时钟振荡器频率下,可以将其时钟源切换到 Timer 1 振荡器。因此,当 CPU 需要全速运行时,可以将时钟源切换到主振荡器。如果某些时间段只有极少的活动,那么就可以慢速运行。此时,可以将其切换到 Timer 1 振荡器,以较低频率运行。12.9.4 节对此已经进行了讲解。这种机制可以有效地降低能耗。在此类微控制器中,还有一个内部振荡器模块,它可以提供更多的时钟源。运行模式的改变由程序代码进行控制,因此在程序运行中可以完成所有切换。

2. 多种空闲模式

383

不难想象,某些情况下有必要保持外围设备运行,同时允许 CPU 停止运行。这类情况仅仅发生在休眠模式的个别情况下,但却是空闲模式的基础。进入空闲模式需要预先设置空闲使能特殊功能寄存器位 **IDLEN**,然后执行 **sleep** 指令。此时将进入空闲模式,执行 **sleep** 之前使用的时钟源将被保留。不同的空闲模式对应于不同的时钟源。如果以较低的时钟源运行空闲模式,就可以极大地降低能耗。

13.9.2 扩展指令集

18F2420 具有常规的 18 系列指令集,包括 75 条指令,这在第 12 章中已经见过。此外,它还具有一个可选指令集,包括 8 条附加指令。这需要在配置设置中对 **XINST** 置位来启用。如果使用了这些指令,就称微控制器运行于*扩展模式*。

附加指令的目的在于增强C编译器的效率。因此,程序员不大可能在汇编程序中使用这些指令,因此这里没有将其列出。它们都与微控制器执行间接寻址和变址寻址(indexed addressing)的能力有关,因此能够提高编译器处理软件栈和其他特性的能力。

此处特别提到了扩展指令集,因为后面将会用到的C编译器C18与此有关。

13.9.3 增强型外围设备

该子型号中的某些外围设备具有“增强”形式。下面将对此做简要总结。

1. 增强型 CCP 模块

这类模块仅存在于大型器件 18F4420 和 18F4520 中。它们具有 13.4 节所述的 CCP 基本特性,但还包含有 4 个 PWM 输出和“翻转”输出极性的能力,也即将激活电平设置为低电平。

2. 增强型可寻址 USART

这种串行通信模块提供了很多重要升级,包括自动波特率检测、自动唤醒模式以及一些其他特性,使其适用于局域互联网(Local Interconnect Network,LIN)总线。

3. ADC 模块

此类 ADC 模块为有 28 个引脚的型号提供了 10 个输入,为有 40 个或 44 个引脚的型号提供了 13 个输入。结合 CCP 模块上的改变,它可以以最小的程序交互建立重复的采集序列。

384

小结

- ❑ 18 系列微控制器沿袭了 16 系列的外围设备,并进行了升级。
- ❑ 这种沿袭始终处于 18 系列体系结构中,尤其要注意的是寄存器映射和中断结构上所发生的变化。
- ❑ 此外,某些外围设备完全沿用了 16 系列的对应模块,其使用方法与之完全相同。
- ❑ 其他外围设备在向 18 系列转变的过程中进行了升级,从 16 系列转向 18 系列的设计员需要查看其中的改变以及它们所产生的影响。
- ❑ 一般而言,将 18 系列外围设备在上电时的默认模式尽可能设计得与 16 系列器件的接近。
- ❑ 在 18F2420 及与其类似器件中,提供了一些 18 系列的增强特性。其中最重要的是“纳瓦”能力和扩展指令集。

参考文献

13.1. PIC18F2420/2520/4420/4520 Data Sheet. (2004). Microchip Technology Inc., DS39631A.

385

第 14 章 C 语言入门

近几章所编写的程序越来越复杂，使用汇编语言来编程也渐趋困难。这主要体现在：程序复杂而难以管理，程序错误难以查找，程序流不易控制，即便是相当简单的数学任务（例如在测光程序中的缩放）也需要耗费很多精力才能实现。

解决上述问题的一种可行的方法是变更编程策略。图 4-1 画出了程序员的困境，同时讨论了 3 种可选的解决方式。本书最初选择第 3 种方式——采用汇编语言编程，因为使用这种程序编写方式可以非常直接地对系统硬件加以控制。但是，基于刚才讨论的那些问题，此时应当考察一下其他的选择。最初高级语言的发明是为了应对程序复杂性并简化调试过程。是否可以在嵌入式环境下应用一种高级语言呢？答案是肯定的，现在我们将选择 C 语言，因为它有出色的能力，能达到我们的目的。

本章以及后续 3 章将首先对 C 语言的基本原理进行介绍，然后结合嵌入式环境下的应用对它所具有的关键特性进行讲解。对 C 语言的讲解并非面面俱到，特别是一些更高级的特性并未涉及。

Microchip 公司的 C18 编译器工作于 MPLAB® 环境下，下面将以此作为学习工具。它可以通过购买获得，也可以从 Microchip 公司网站下载免费的学生版。本书附属资源中包含这个编译器。本书假定读者已经获得此编译器。

下面的学习将分层次进行。首先，将对 C 语言本身进行介绍，这部分知识与嵌入式系统无关，可以应用于台式计算机和其他平台。与此同时，将讲解如何在嵌入式系统中应用 C 语言，尤其是在 PIC® 环境下。为了使上述两条线索同时进行，本章前面部分会对 C18 编译器进行介绍。然后将通过循序渐进的实例进行 C 程序的编写实践，部分例程用于 Derbot-18 AGV。在讲解 C 语言的过程中，不用很长时间就能够体会到它与汇编语言相比之下的优势。

在本章结束时，你会学到以下内容：

- ☐ C 语言的核心特性；
- ☐ C 编译器 MPLAB C18 及其库文件的核心特性。

如果你对 C 语言已经非常熟悉，可以跳过本章的前 3 节，这不会影响对后续几节内容的阅读。

14.1 为何选择 C 语言

4.1 节简单介绍了高级语言（high-level language，HLL）的概念。HLL 是一种采用

了便于人类理解的语言与结构的编程语言。与此同时，它具有清晰定义的规则，因此所编写的程序能够精确地转换成机器码。

无论使用哪种 HLL，都需要有一个中间程序将所编写的程序转换为计算机机器码。如果在程序执行之前对程序进行这种转换，那么这种转换程序被称为编译器(compiler)。例如，C 语言就使用编译器。如果程序一边执行一边转换，那么将其称为解释器(interpreter)。例如，Basic 语言就使用解释器。

HLL 的一个显著优点在于它的轻便性。程序员编写的程序与它所运行的计算机平台无关。编译器或解释器则与特定的机器有关，负责创建实际的机器码。这样，相同的源代码就可以(理论上——必须反复强调这一点)运行于完全不同的计算机上。这自然与完全取决于目标计算机的汇编方式完全不同。

在嵌入式应用中使用 HLL 时，自然希望它能够带来更简单更可靠的代码编写过程，但同时又希望它能够尽可能多地保留汇编编程中与硬件的紧密接触。C 程序最初是为 20 世纪 70 年代更为简单的计算机所编写的。尽管在台式计算机和工作站点的应用中 C 语言已经被很多其他语言所赶超，但是在与计算机硬件密切相关的工作中(如嵌入式系统中)，它依然是一种功能强大的语言。它不仅具有我们所需要的所有 HLL 的特性，同时也允许对硬件元素进行访问。

14.2 C 语言简介

尽管从今天来看，C 语言是一种相当简单的语言，但它同样可以编写出复杂而成熟的程序。本节将利用一个简单的例程给出 C 语言的基本要素，目的在于读者能够尽快转向使用 MPLAB C18 编译器，并开始为 PIC 18 系列微控制器编写简单的程序。

14.2.1 简史

C 语言诞生于 20 世纪 70 年代末在美国新泽西州的贝尔实验室，最初的公开发行文档于 1978 年由 Brian Kernighan 和 Dennis Ritchie 所发布。该版本的影响非常广泛，因而通常被称为 K&R 版本。1989 年美国国家标准学会(ANSI)采纳了一个 C 语言版本，将其确定为标准 X3.159-1989。认识这一标准非常重要，因为 Microchip C18 编译器即基于此标准。此标准得到了广泛的认可和采用，很多都以 ANSI C 为参考。1990 年，国际标准化组织(ISO)采纳了同一版本作为国际标准，并在 1995 年和 1999 年进行了修订。1999 年的版本中所包含的扩展内容，在很多用于嵌入式系统的编译器中并没有实现。

14.2.2 第一个 C 程序

讲解 C 语言的书一般都会以向屏幕输出字符 Hello world 的例程作为开始，这已形成惯例。然而在很多嵌入式系统中并不存在这样的屏幕以供写入。作为替代，在几

乎所有嵌入式系统中都要求能够向某个端口输出数据。因此，在第一个例程中，将仅对某个数进行递增操作，然后将其写入 18 系列微控制器的端口 B，如例程 14-1 所示。

例程 14-1 递增端口 B 的输出值

```
/*****************************************************************************
Example1.
Introductory Example of C Programming, with PIC 18 Series Microcontroller.
8-bit value output by Port B is continuously incremented.
Files c018i.o and p18f242.lib are included by the Linker Script.
TJW 21.10.05                                               Tested 23.10.05
*****************************************************************************/

//Include 18F242 header file, for all processor-specific declarations
#include <p18f242.h>

unsigned char counter;      //specify counter as unsigned character

void main (void)            //main function starts here
{
  TRISB = 0;                // initialise all bits of PORTB as output
  counter = 1;              //counter value is initialised to 1

  while (1)
    {
    PORTB = counter;        // Move 'counter' value to Port B
    counter = counter + 1; //Increment counter
    }
}
```

14.2.3 程序结构——声明、语句、注释和空格

C 语言是一种所谓的形式自由的编程语言。也就是说，程序不必遵循某种严格的结构形式。这与汇编语言不同，在汇编语言中，一行中某个命令字的位置有时是很重要的。

粗略地讲，C 程序由以下几个部分组成：*声明*（declarations），用于设置程序背景和初始化变量；*语句*（statements），程序在这里执行；*注释*（comments），对程序的执行内容提供注解供其他人员阅读；*空格*（space），在所用字符与符号之间留有一些必要的间隙，从而增强整个程序结构的可读性。下面将对它们逐个进行讲解。

1. 注释

注释以字符组合/* 开始，以*/结束。这种形式允许将注释分多行放置，同时也可以将注释放在声明或语句的前面或后面。

在上述例程中可以看到，程序起始处的 7 行都由注释组成，这些注释部分通过星号行组合在一起。在这个注释块下面是一个空行。这在 C 语言中是允许的，使用空行有助于理解程序。

此外，另一种注释形式是在注释前加上双划线//。这种注释只能维持到双划线所在行的结尾，并且不需要终止符。同时使用两种注释方式可以带来很多便利，前者可用于主要的注释块，后者则用于单行注释。本书也采用了这种方式，在上述例程中也可以看到。

对于汇编语言的编程，灵活使用注释同样重要，这样可以清晰简洁地说明程序的执行内容。

2. 声明

声明的使用有多种方式，用于创建程序元素（如变量和函数）并说明特性。声明非常重要，因为C语言中的所有变量和函数在使用之前必须首先进行声明。需要指定的特性包括数据元素的类型（例如定点还是浮点）、存储地址或函数特性。声明用分号终止。

在例程中，代码行

```
unsigned char counter;     //specify counter as unsigned character
```

就是声明，其含义在后面将简单进行介绍。

在简单程序中，声明一般出现在程序的开头，这看来是很合理的。但在一些更为复杂的程序中，声明可能出现在程序内部，声明出现在不同位置表示不同的含义。

3. 语句

语句是程序执行的地方。它们用于执行算术或逻辑操作，并建立程序流。不构成代码块的每条语句（如下所示）都以分号结束。

语句有多种不同类型。最一般的语句是*表达式*（expression），其中包含了算术操作。例程14-1中的表达式语句如下：

```
TRISB = 0;        //initialise all bits of PORTB as output
counter = 1;      //counter value is initialised to 1
```

除了程序分支之外，语句将按照它们在程序中的出现顺序进行执行。

389

4. 代码块

声明和语句可以放在一起构成*代码块*（code block）。代码块包含在大括号（花括号）中。例程14-1中的代码块如下：

```
while (1)
{
 PORTB = counter;        //Move 'counter' value to Port B
 counter = counter + 1; //Increment counter
}
```

代码块还可以放在其他块中，每个块都有自己的括号对。在编写C程序时跟踪这些括号也许算得上是一项很重要的“消遣”，因为在某些复杂程序中，也许会有很多代码块嵌套在一起。通常将一页中相互匹配的括号对在垂直方向上保持呼应，同时每嵌套一个括号对都向右缩进一段距离。这样就可以清晰地跟踪每个括号对。

5. 空格

在C程序中灵活使用空格可以有效地提高程序的清晰度。在字符之间需要用空格隔开，从而避免合成一个字符，例如上面所引用的声明示例。其他形式的空格(包括空行)会被编译器所忽略，程序员可以利用空格来优化程序结构。这不仅适用于空行，同样适用于代码行的缩进。例如，在上面的代码块示例中，括号分别占据一行，并且在垂直方向上相互呼应。如果按照下述方式编写程序，会产生相同的编译结果：

```
while (1){PORTB = counter; // Move 'counter' value to Port B
counter = counter + 1;}     //Increment counter
```

但是，上述代码阅读起来不够清晰，特别是代码块的结尾位置不够清楚。当程序复杂度上升时，对良好的程序结构的需求也随之上升。

14.2.4 C语言关键字

C语言中只有32个关键字。它们列写在表A6-1和表A6-3中，并附带简单描述。从中可以看出，大多关键字都与数据类型有关。在示例程序中，声明：

```
unsigned char counter;     //specify counter as unsigned character
```

声明了一个名为**counter**的变量，并使用关键字**unsigned char**将其指定为无符号字符类型。

其他关键字与程序流有关(见表A6-2)。上述示例中的**while**关键字建立了一个连续循环，相关描述可查阅14.2.8节。

关键字可以被编译器所识别，在使用时应符合规定的上下文。它们不能用于其他目的，例如不能作为变量名。

390

14.2.5 C语言函数

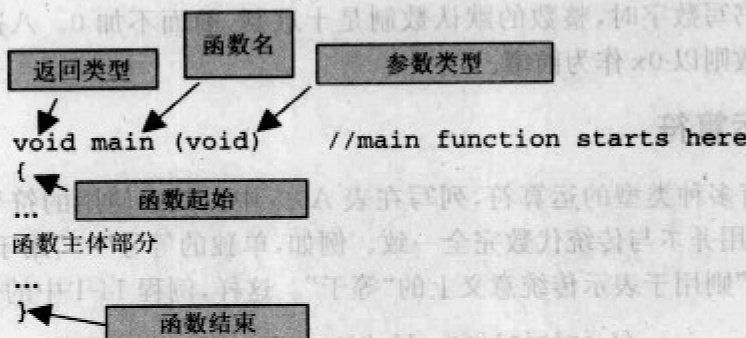
C程序的基本结构是*函数*(function)。每个程序至少应有1个函数，称为*主函数*(main)。程序从这个函数开始执行，并且包含在其中。

除了主程序之外，某些函数与汇编语言中的子例程有些类似。它们的使用方法与之类似，通常包含一个明确的程序动作。良好的程序结构倾向于将程序的大部分放置在多个函数中，然后在主程序中调用这些附属函数。函数之间可以相互调用。

C函数与汇编子例程的区别在于，在调用程序与被调用函数之间的数据传递分别具有不同的控制机制。可以向函数传递一种数据元素，称为*实参*(argument)。但是，这些实参的数据类型必须与先前声明的类型一致。只允许存在一个返回变量，其数据类型也是事先声明的。向变量传递的数据仅仅只是原变量的副本。因此，函数并没有改变所指定的变量。这样，函数的影响是可以预测并且可以控制的。通常使用*形式参数*(terminology parameter)来替代实参。这二者之间的区别在C语言中有详细的描述。但在这几章中暂不对其加以区分。

在程序中，函数的定义通过一种具有特定属性的代码块来实现，其中第1行是函

数头。下面给出了例程中的函数头，描述了函数的一般形式。



首先要给出返回类型。本例中，使用关键字 **void** 表明没有返回值。对于 **main** 函数而言，这是一般的做法——毕竟，它不需要向某个地方传递数值。在函数名之后的括号内列出了 1 个或多个数据类型，用于明确必须向函数传递的参数。本例中没有参数需要传递（对于 **main** 而言这是一般的做法），因此再次使用了 **void**。

在函数头之后是一对括号，其中的代码构成了函数自身。函数中可以包含任何内容，长度从 1 行到若干页。函数的最后一条语句可以是 **return**，用于说明向调用程序返回的数值。这并非必要，有时可能不需要返回值。

从例程可以看出，为了表达清晰，程序中包含 **main** 函数的括号紧靠在左边，而包含 **while** 语句的括号则进行了缩进。

391

在例程 14-1 中，只包含一个 **main** 函数。在后续章节中将会看到，当使用了多个函数时会产生一些其他问题。

14.2.6 数据类型与存储

C 程序中的变量拥有 4 个属性：名称、类型、取值和存储地址。上面已经提到，在变量使用之前必须声明数据类型（例如有符号还是无符号，定点数还是浮点数）。一旦指明了变量类型，编译器就可以确定存储变量所需的存储器空间大小。然后，就可以在必要时对变量进行初始化。

表 A6-1 列出了定义数据类型时使用的字符。每种数据类型所需的实际存储空间会随不同的编译器而变化。表 A6-4 列出了 MPLAB C18 编译器中的可用数据类型以及所需的存储空间。例如，例程 14-1 中的变量 **counter** 被定义为无符号字符。表 A6-4 显示 C18 编译器将为其分配一个 8 位的存储地址。虽然类型显示它必须是一个字符，但实际上它可以用于存储任何 8 位的数字。这是一种很有用的数据类型，因为在 PIC 环境下会用到很多单字节变量。下面将会看到，**PORTB** 和 **TRISB** 也被定义为无符号字符。

数据名称必须以字母开头。在编写复杂的程序时，一般的做法是在名称开头用 1 个或几个字母来标明变量类型，例如例程中的变量名 **counter** 可以改成 **uicounter**，用于

提醒程序员这是一个无符号整型数。这样可以帮助程序员记住数据类型从而减少编程错误。但在本书中,并没有采用这种做法。

在程序中书写数字时,整数的默认数制是十进制,前面不加 0。八进制数用 0 开头。十六进制数则以 0x 作为前缀。

14.2.7　C 运算符

C 语言具有多种类型的运算符,列写在表 A6-5 中。其中使用的符号是非常熟悉的,但是具体应用并不与传统代数完全一致。例如,单独的等号"="用于向变量赋值,两个等号"=="则用于表示传统意义上的"等于"。这样,例程 14-1 中的代码行:

```
TRISB = 0;        // initialise all bits of PORTB as output
```

表示将名为 **TRISB** 的变量赋值为 0,可以读作"变量 **TRISB** 取值为 0"。在 18F242 的头文件中已经将 **TRISB** 定义为无符号字符,也即 8 位数字。根据端口特性,上述代码行将端口 B 的所有位均设置为输出。

如表中所示,运算符具有不同的执行优先级。编译器在分析语句时将应用这一顺序。如果在一条语句中出现多个具有相同优先级的运算符,那么将按顺序从左至右或从右至左执行,如表 A6-5 所示。

392

例程是一个很简单的程序,其中的代码行:

```
counter = counter + 1;
```

包含 2 个运算符。表 A6-5 显示加运算符的优先级为 4,赋值运算符的优先级为 14。于是加运算将首先执行,然后是赋值运算。最终结果是变量 **counter** 增加 1。

14.2.8　程序流的控制以及 while 关键字

表 A6-2 中的所有关键字都与程序流有关,例如程序的循环和分支。在上述首个例程中使用了 **while** 关键字,只要满足特定条件就可以循环执行某条语句或语句块。后面还将遇到很多类似的分支和循环结构。

一般的 **while** 结构是:

```
while (conditional expression)
statement;
```

只要条件表达式结果为真(即非零),**statement** 部分将反复执行。一旦条件为假,将执行循环体之后的程序。

如果有多条语句需要与 **while** 相关联,那么可以在花括号中包含一系列语句:

```
while (conditional expression)
{statement 1;
statement 2;
statement 3;
}
```

注意,对 **while** 条件的分析是在循环执行起始处进行的,如果为真将执行整个循环

393

体，即便在循环体执行过程中条件发生了改变也依然如此。

在例程中，通过在条件表达式中放置"1"来强制执行连续的循环。

```
while (1)
    {
    PORTB = counter;        // Move 'counter' value to Port B
    counter = counter + 1;
    }
```

在 **while** 括号内的两条语句将无限重复下去。

14.2.9 C 预处理器及其伪指令

编译过程由若干不同的阶段构成。其中的第一步是由预处理器(preprocessor)来完成的。它将对所能找到的所有伪指令(directive)做出反应。它们的行为与汇编语言的指令类似，向编译器自身传递指令。表 A6-6 列出了伪指令的例子。伪指令的格式要求每条指令占据一行。行末不需要添加分号。

例程中的代码行：

```
#include <p18f242.h>     //for all 18F242 declarations
```

使用了 # **include** 指令来包含一个处理器相关的头文件。该头文件适用于 C18 编译器，其中包含了特定的 18 系列处理器所需的必要声明，这样就无需在源代码中重新录入。文件中包含有所有 SFR(特殊功能寄存器)的声明，包括程序中用到的端口 B 所对应的 SFR。

14.2.10 使用库和标准库

由于 C 语言是一种简单的语言，它的功能大多来自于一些标准函数和宏，这些函数和宏在任何编译器所附带的库中都是可用的。C 库中包含有一系列预先编译好的函数，以目标文件的形式存在，可以连接到具体应用中。在 ANSI 标准中对标准库(Standard Library)的内容进行了定义。其中包括用于输入/输出的函数、大量数学运算函数(如所有三角函数)以及其他数据处理函数。

后面将会看到，除了标准库之外，编译器还可能具有它自己的函数库，专门用于其目标环境。

14.3 编译 C 程序

在完成 C 源代码之后，将对其进行编译。最终产生的文件中包含有与源码等价的机器码，可以由目标机直接执行。

上述已经提到头文件和库文件的概念。它们已经得到了广泛的应用，几乎每个 C 源文件都不是孤立的。一旦有"额外"的文件加入，最终的可执行程序将由多个相关文件来构建，方式通常非常复杂。同样地，编译 C 程序的过程将产生多个输出文件。

图 14-1 概略地描述了编译的一般过程，包括用到的文件以及生成的文件。主程序（C 源文件）以 C 语言编写，放在扩展名为 .c 的文件中。其中一般都会包含其他标准文件（使用伪指令 #**include**），例如刚才已经认识的处理器相关头文件。一个源文件以及它所包含的所有头文件被称为一个*翻译单元*（translation unit）。

394

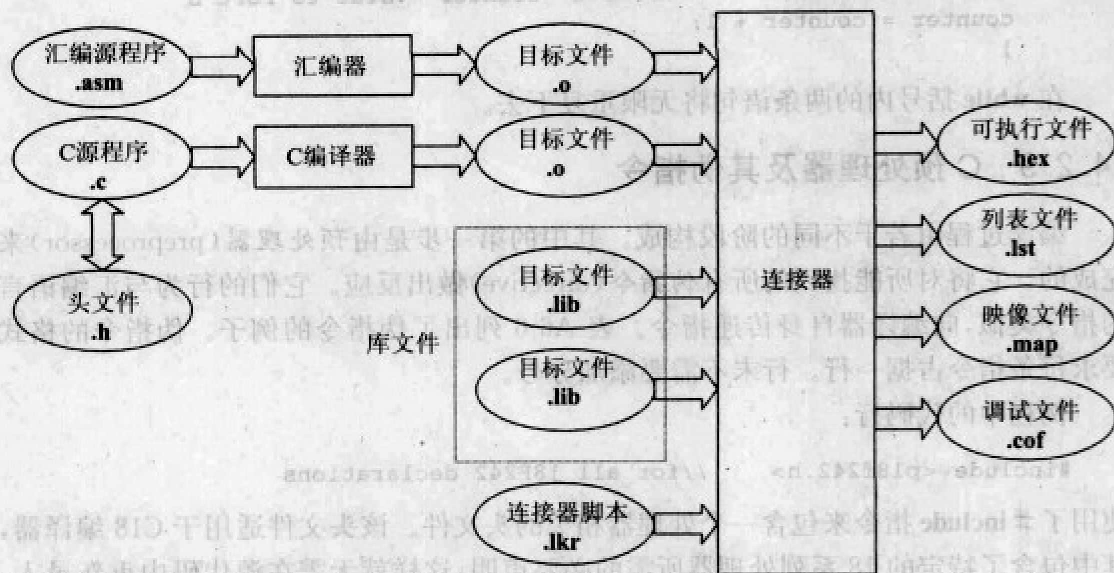


图 14-1　C 文件结构以及 C18 文件扩展名

源程序编写完成之后，使用 C 编译器进行编译生成*目标文件*。这种文件由可重定位代码构成，并不完全映射到处理器存储器映射中。其他文件也得到进一步处理（包括汇编语言源文件），以类似方式进行编译或汇编，最终生成以可重定位代码为主要内容的目标文件。

在这个阶段，除最简单的程序之外，多数情况下需要与其他文件进行组合，这些文件都已成为目标文件。这些目标文件可能来自于编译器相关库，或者由程序员或公司事先生成。下面的任务交给*连接器*，它负责将不同的文件组合在一起并构建一个单独的可执行文件。这个过程由*连接器脚本*（Linker Script）指导完成，连接器脚本用于定义处理器存储器映射并提供其他信息，并有可能生成预编译文件。在例程 14-1 的起始标题块中，提到两个目标文件 **c018i.o** 和 **p18f242.lib**，它们将由连接器连接到程序中。第 17 章将对其中的具体过程以及目标文件所扮演的角色进行解释。

连接过程完成之后，如果没有错误，将产生一系列输出文件。14.5.4 节将对 Microchip 公司 C18 编译器的相关细节进行讲解。

14.4 MPLAB C18 编译器

MPLAB C18 编译器是 Microchip 公司自己的 C 编译器，专门为 PIC 18 系列微控

制器编写。它遵循 ANSI X3.159-1989 标准,在此基础上还添加了一些扩展,在设计上针对 PIC 微控制器做了优化。

C18 编译器运行于 MPLAB IDE 主环境下,与汇编器、连接器和库协同工作。一旦安装,编译器将被连接到这个工具组件中。与 MPLAB 不同,编译器需要购买,价格在 $150 左右。在编写本书时,已经有一个免费的学生版本可供使用,可以从 Microchip 公司网站下载,在本书附属资源中也可以找到。

关于此编译器已有一些很好的文档,可以查阅参考文献 14.1 和参考文献 14.2。这些文档也可以从 Microchip 网站下载。下面将对编译器进行简单的介绍,这些知识对于运行给定的所有例程已经足够,要了解更多细节可以查阅参考文献。

14.4.1 数制规范

MPLAB C18 可以识别出 14.2.6 节所提到的 C 语言的数制约定。另外,还可以使用前缀"0b"用于标记二进制数字。例如:

```
TRISA = 0b'10000110';     // initialise PORTA
TRISB = 0x86;             // initialise PORTB
TRISC = 134;              // initialise PORTC
```

向每个 TRIS 寄存器送入相同的值。

14.4.2 算术运算

ISO/ANSI C 标准要求所有的算术运算以整型(即 16 位)或更高的精度完成。C18 编译器服务于 8 位环境,因此并没有遵守这个规定,而是以 **char** 数据类型执行算术运算。

本书后续章节假定读者已经安装了 MPLAB IDE 和 MPLAB C18 编译器,并遵循参考文献 14.1 给出的简单步骤。所使用的 C18 编译器的版本为 3.00。

14.5 C18 指南

以上所述的程序看似简单而可信,下面不妨尝试进行编译和仿真。在 MPLAB 中打开新项目并赋予一个合适的名称;下图中使用了一个常用名称 **example1**。打开新文件并输入例程 14-1 的代码。使用某个文件名(如 **example1.c**)进行保存,然后使用 Project > Add Files to Project 将该文件添加到项目中。

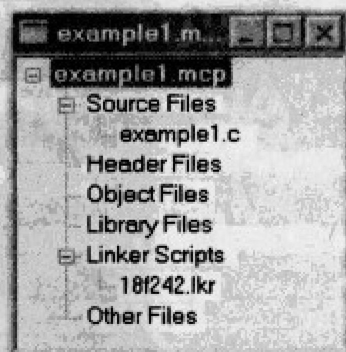


图 14-2 加入连接器脚本

14.5.1 连接器和连接器脚本

由于编译过程将自动使用连接器(Linker),因此在项目中包含连接器脚本(Linker Script)是非常重要的。第 17 章将对它的格式进行具体讲解,这里只需加以指定即可。如图 14-2 所示,在项目窗

396 口中右击 **Linker Scripts** 行。此时将出现 **Add File** 快捷菜单。单击菜单，通过 **Add Files to Project** 窗口寻找文件夹 **mcc18/lkr/**。在文件夹中单击 **18f242. lkr** 文件。此时项目窗口将如图 14-2 所示。

注意，这里在调用连接器脚本时使用了它的原始版本和地址。对于更高级的项目，最好复制该文件并将副本放置在项目目录下。这样就可以对文件进行修改而不损坏原文件。

14.5.2 连接头文件和库文件

在例程中，通过下列代码行调用头文件：

```
#include <p18f242.h>
```

因此，必须确保编译器能够识别出该文件的搜寻路径。此外，连接器脚本所调用的其他库文件同样要求能够找到。使用 **Project > Build Options > Project > General** 可以查看这些路径设置是否正确。此时将弹出图 14-3 所示的窗口。如果编译器被安装在默认地址中，只需单击 **Suite Defaults** 即可，它将选择图中列出的路径，然后单击 OK 按钮。

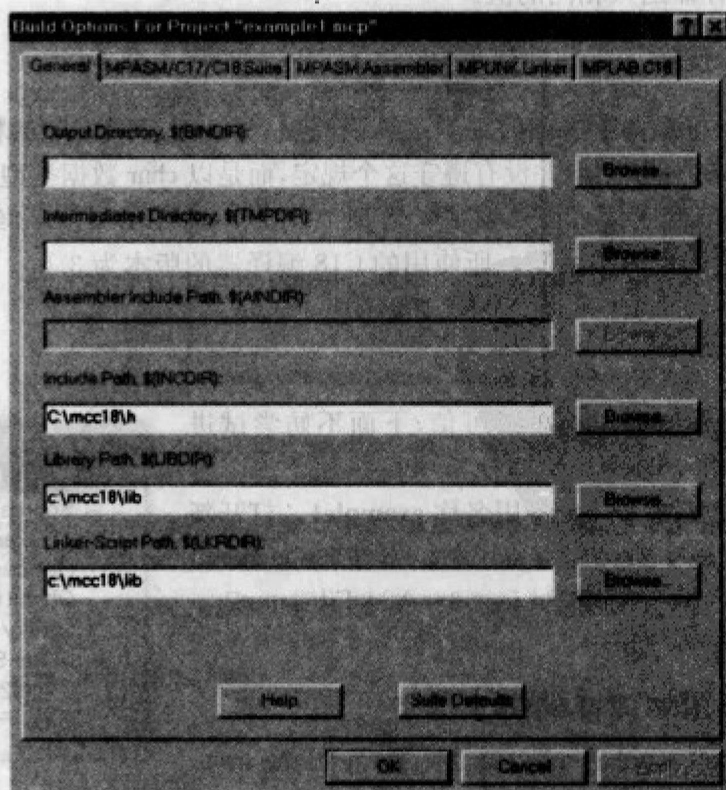


图 14-3 设置头文件的搜索路径

14.5.3 构建项目

此时就可以构建项目。与汇编器类似，单击 **Project > Build All**。如果所有内容输入无误并且连接正确，最终将在输出窗口中显示 **Build Succeeded** 信息，如图 14-4 所示。注意构建过程所需的时间要比汇编器长的多，即便是对于类似例程的小程序。如图 14-4 所示，这个过程包含若干个不同的阶段。在编译阶段，窗口中显示出若干编译选项(标记为-Ou-，-Ot-，-Ob-等)正在被执行。它们是编译器的默认设置，在使用编译器的初级阶段无需了解更多的细节。

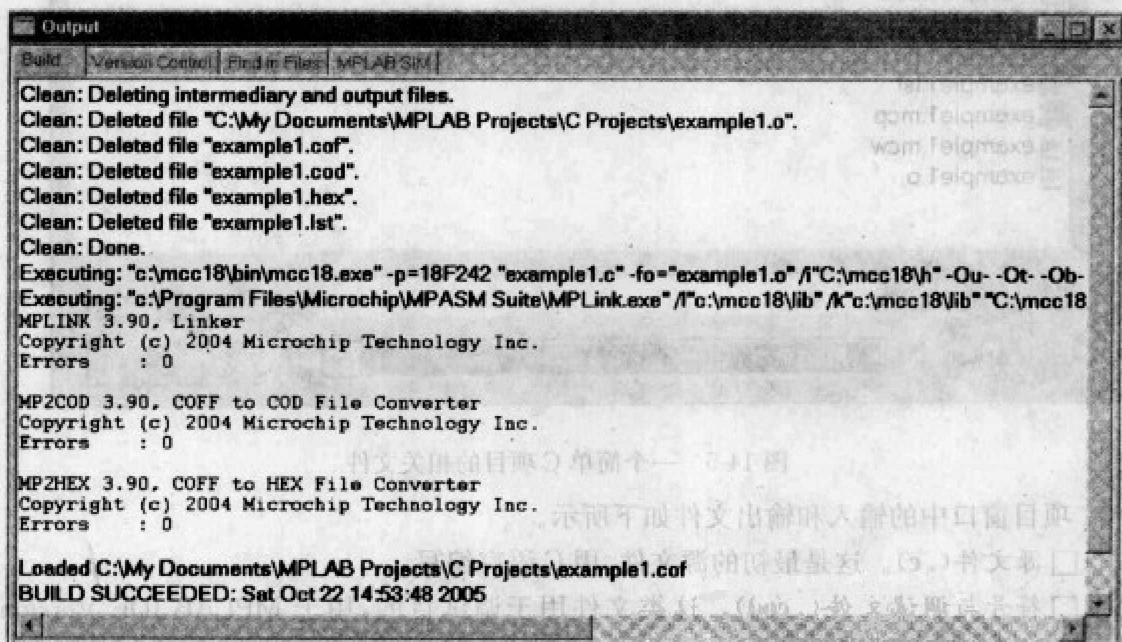


图 14-4 构建成功之后的输出窗口

如果项目没有构建成功，有必要仔细查看结果中的出错信息。编译器将首先检查程序的语法是否正确，也即程序的编写是否符合 C 语言的格式要求。即便结果中未显示出任何错误信息，也可以尝试去掉某个分号或加入一些其他的小错误，再检查编译器在构建过程中的反应。如果存在语法错误，编译器将在输出窗口中进行报告，并给出出错误的代码行号。在 MPLAB 窗口中，可以使用 **Edit > Properties > Editor** 并单击 **Line Numbers** 框来开启代码行。需要注意的是，编译器所给出的出错代码行号可能并不是真正需要纠正的位置。举个简单的例子，如果将 **main** 的起始括号"注释掉"，将会发现最终报告出的语法错误在下面一行——事实上这一行 C 代码是完全正确的。

当清除所有语法错误之后，编译器将检查其他类型的错误并报告出错参考号。参考文献 14.2 列出了这些出错号，并给出了简要的出错说明和可能的解决途径。

14.5.4 项目文件

项目构建成功之后将产生如图 14-5 所示的文件。其中有很多文件都可以在图 14-1中看到。

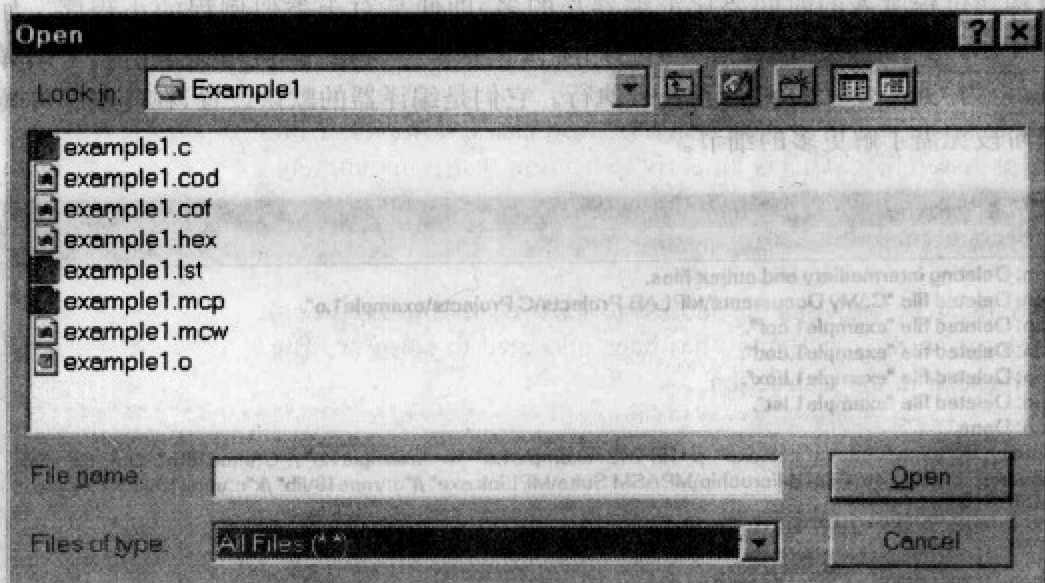


图 14-5 一个简单 C 项目的相关文件

项目窗口中的输入和输出文件如下所示。

- ☐ 源文件(.c)。这是最初的源文件,用 C 语言编写。
- ☐ 符号与调试文件(.cod)。这类文件用于调试目的,用于 MPLAB IDE Version 5.xx 及更早的版本。
- ☐ COFF 目标模块文件(.cof)。这种文件用于提供调试信息,用于 MPLAB IDE V6.xx 及更新版本。
- ☐ 可执行文件(.hex)。这是实际的程序代码,可以下载至微控制器,用于模拟或仿真目的。
- ☐ 列表文件(.lst)。此类文件给出了与目标代码紧邻的初始源代码,另外还提供有符号值、存储器使用信息以及出错及报警信息。
- ☐ 目标文件(.o)。这种文件包含有可重定位代码。它们是编译器或汇编器的输出,并构成连接器的输入。在库中也可以找到目标文件。

14.6 仿真 C 程序

下面将对例程 1 进行仿真。程序本身非常简单,从程序的仿真过程中可以了解 C 程序的构建过程,更加深对图 14-1 的认识。

如 4.7 节所述，选择 **Debugger > Select Tool > MPLAB SIM** 可以使用 MPLAB 仿真器。单击 **View > Watch** 打开观察窗口，查看 **counter**、**PORTB** 和 **PCL** 的值。使用调试器工具条（见图 4-9），复位程序计数器，注意程序计数器（**PCL**）将按要求清零。但是，此时将自动弹出另一个程序窗口。这是 **c018i.c**，是连接器所调用的启动程序。其中包含某种处理器初始化程序，用于 C 程序的正确执行。在第 17 章中将对此进行进一步讨论。

单步执行程序（或者使用 Animate），此时将看到执行动作最终显示在源程序中。在观察窗口中可以看到 **counter** 和 **PORTB** 的值在递增。此时，程序执行已经停留在 **while** 循环中。

如果要查看 C 程序的构建过程，可以选择 **View > Disassembly Listing** 浏览反汇编列表。在此窗口中（如图 14-6 所示）可以同时查看原始 C 代码和相应的汇编代码。从中可以看出，有几行 C 代码被翻译为单行汇编语言。但多数情况下，单行 C 语言必须替换为多行汇编语言。这也大致说明了用 C 语言来编程可以获得更简单的源文件。随着更复杂的 C 结构的引入，C 代码与其反汇编列表的行数之比会变得更为显著。

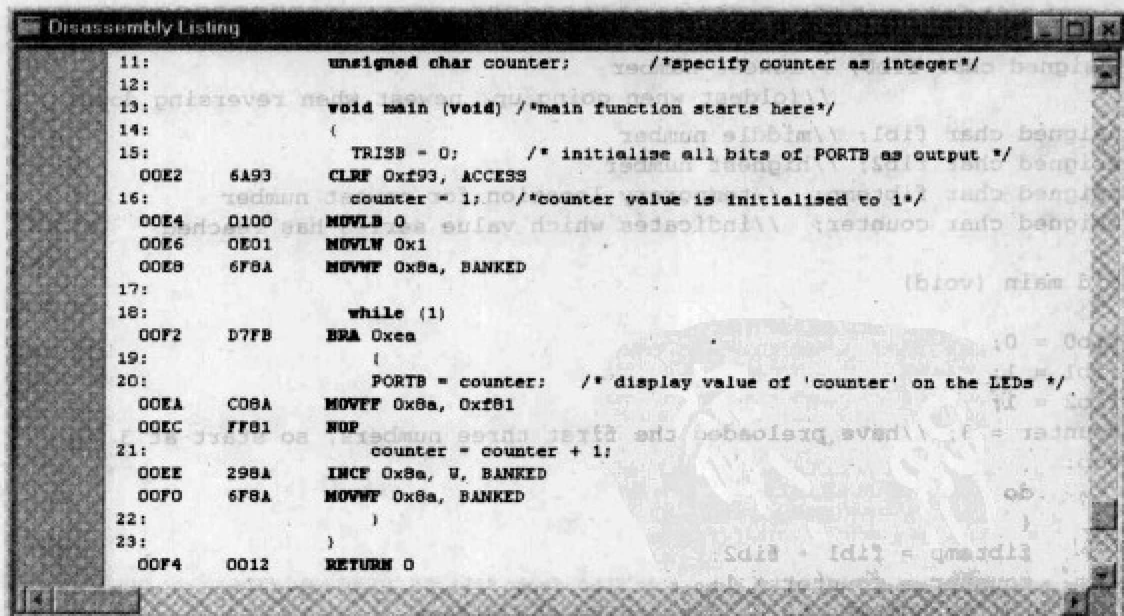


图 14-6　例程 14-1 的部分程序所对应的反汇编列表

如图 14-6 所示，对于 **TRISB** 给出了正确的地址 $0F93_H$（见图 12-5）。存储区 Bank0 中的存储地址 $8A_H$ 已经分配给了 **counter**。下列指令将 **counter** 中的值传送至 **PORTB**。

```
MOVFF 0x8a,0xf81
```

其中 0xf81 是端口 B 的地址。类似地，可以查看 counter 的递增过程。注意，这些用存储地址写出的汇编指令并没有按正常顺序放置。存储地址 $00F2_H$ 处的分支指令事实上用于形成持续循环，因此应放在列表结尾。

399 ~ 400

14.7 第2个C例程——斐波那契程序

下面将进一步学习C程序，给出一个稍微复杂点的程序。例程14-2给出了我们并不陌生的斐波那契序列产生器程序的C版本。这个程序用于计算斐波那契序列，首先增长至某个量级，然后再回退，整个过程不停地循环往复。

例程14-2 斐波那契序列产生器

```
/************************************************************************
Fibonacci
In a Fibonacci series each number is the sum of the two previous numbers.
This program calculates Fibonacci numbers within an 8-bit range,
Files c018i.o and p18f242.lib are included by the Linker Script.
Program intended for simulation only, hence no input/output.
TJW 21.10.05                                           Tested 23.10.05
;***********************************************************************/

#include <p18f242.h>

//these memory locations hold the Fibonacci series
unsigned char fib0; //lowest number
                    //(oldest when going up, newest when reversing down)
unsigned char fib1; //middle number
unsigned char fib2; //highest number
unsigned char fibtemp;  //temporary location for newest number
unsigned char counter;  //indicates which value series has reached

void main (void)
{
 fib0 = 0;
 fib1 = 1;
 fib2 = 1;
 counter = 3; //have preloaded the first three numbers, so start at 3
loop:
      do
      {
      fibtemp = fib1 + fib2;
      counter = counter + 1;
//now shuffle numbers held, discarding the oldest
      fib0 = fib1; //first move middle number, to overwrite oldest
      fib1 = fib2;
      fib2 = fibtemp;
      }
      while (counter<12);
//when reversing down, we will subtract fib0 from fib1 to form new fib0
      do
      {
      fibtemp = fib1 - fib0; //latest number now placed in fibtemp
      counter = counter - 1;
//now shuffle numbers held, discarding the oldest
      fib2 = fib1; //first move middle number, to overwrite oldest
```

```
            fib1 = fib0;
            fib0 = fibtemp;
            }
            while (fib0>0);
            goto loop;
    }
```

14.7.1 程序初步——进一步认识变量声明

在起始注释之后，程序首先声明了5个无符号字符变量，其中3个变量在 **main** 开始处进行了初始化。还有一些方法可以简化这一过程。首先，一种数据类型的声明不必只局限于单个变量。因此，5个变量可以一起声明：

```
unsigned char fib0, fib1, fib2, fibtemp, counter;
```

程序员可以选择是否采用这种方式。其中的缺点是，当需要为每个变量添加注释时可能会有点不太方便。

另外，还可以在变量声明时进行初始化。可以采用下述方式：

```
unsigned char fib0 = 0;     //lowest number
unsigned char fib1 = 1;     //middle number
unsigned char fib2 = 1;     //highest number
```

同样地，采用这种方式是否有利，取决于程序员。

14.7.2 do-while 结构

在斐波那契程序中有2个循环，它们采用了 **do** 和 **while** 关键字。只要 **while** 条件（第1个循环条件为 **counter** < 12）为"真"（即非零值），关键字 **do** 后面的代码块就总是执行。在这种循环结构中，在检测 **while** 条件之前，do 代码总是至少执行一次。这与 while 循环结构不同，在 while 结构中只要条件不满足，循环体将一次也不会执行。

在序列回退过程中，在 **while** 语句中对 fib0 进行检测，在 fib0 达到 0 之前将反复执行循环。

14.7.3 标号和 goto 关键字

紧挨第1个 **do** 循环之前有一个表达式 **loop:**，这是一个标号，终止符为冒号。程序末尾一行使用了关键字 **goto** 返回到 **loop:** 继续执行。这类标号仅用作 **goto** 指令的目标，没有其他用途。指令 **goto** 可以在给定函数内部实现无条件跳转，不能跳转到其他函数。在 C 语言编程环境下一般不推荐使用 **goto** 分支指令，因为它的滥用会破坏良好的程序结构。

14.7.4 仿真斐波那契程序

与前面的工作类似，此时可以对程序进行仿真。从本书附属资源中将该程序复制到合适的项目中并使用 MPLAB SIM 进行仿真。打开观察窗口并显示出 **PCL** 和所有

变量。单步执行程序，一旦进入 **main** 就可以观察循环是如何被控制的。注意当 **counter** 达到 12 时，do 代码块将执行完毕然后开始执行下面的循环。第 15 章的例程 15-2 给出了该循环的另一种结构。

14.8 MPLAB C18 库

在本章的结尾将介绍一些 C18 编译器中的可用库函数。C18 编译器中包含很多类型不同的库，包括 C 标准库以及与 PIC 微控制器有关的很多函数。使用该编译器编写的多数程序都或多或少地应用了这些函数，每个程序至少会包含一个这样的库函数。本书自此开始给出的所有例程都会使用一些库函数。

403

库参考手册[14.3]对每个库函数都给出了详细的讲解。其中说明了函数功能、需要传递的参数、返回值以及必须包含的头文件。下面将对不同类型的库函数进行讲解。

14.8.1 硬件外围设备函数

这类函数与微控制器外围设备有关，用于启用和配置外围设备、切换运行模式、读出外围设备或禁止外围设备。例如，与 ADC 相关的可用函数显示在表 14-1 中。可以看出，表中给出了在使用 ADC 时通常会用到的所有功能（必要时可以参考 11.3 节）。在库参考手册[14.3]中可以详细查看函数的参数，了解具体设置。在第 16 章的例程中用到了其中的某些函数。

当然，还可以直接通过相应的 SFR 对外围设备进行操作。但通常，使用库函数会使编程过程变得更为简单，也更易于观察，从而使整个过程更为可靠。详细了解 SFR 的细节依然非常重要，某些情况下也必不可少。这对于认识函数参数的设置方法很有帮助。

这些函数仅属于 C18 编译器，在 C 标准库中并不存在。

14.8.2 软件外围设备库

此类函数用于为系统中所包含的一些外部设备提供驱动函数。其中包括日立 HD44780 LCD 驱动器（参加第 8 章），MCP2510 CAN（控制器区域网络，参见第 20 章）接口，以及在软件中产生串行交互的函数。

与硬件外围设备库类似，这类函数也仅属于 C18 编译器，在 C 标准库中并不存在。

表 14-1 C18 中与 ADC 相关的库函数

函　数	功　能
OpenADC()	配置 ADC
SetChanADC()	选择要用的通道

(续)

函　　数	功　　能
ConvertADC()	开始 ADC 转换
BusyADC()	测试 ADC 当前是否正忙
ReadADC()	读取 ADC 转换结果
CloseADC()	禁止 ADC

404

14.8.3 通用软件库

此类库中的函数包括来自 C 标准库中的函数以及与 Microchip 公司相关的函数，分为以下几类。

1. 字符分类

此类函数可以满足标准 C 库 **ctype** 的要求，对字符进行测试以确定其特性。表 14-2给出了示例。

表 14-2　字符分类函数示例

函　　数	功　　能
isalnum()	确定字符是否为字母或数字
isalpha()	确定字符是否为字母
iscntrl()	确定字符是否为控制字符
isdigit()	确定字符是否为十进制数

2. 数据转换

此类函数可以在不同的数据表达格式间进行转换。这在嵌入式环境下是很有用的，例如可以将数据由二进制转换为字符串用于输出，或者从键盘读入字符串并转换为二进制数。在表 14-3 中给出了一些例子。其中既有标准 C 函数，也有 C18 中的"特有"函数。

表 14-3　字符串/字符转换函数示例

函　　数	功　　能
atop()	将字符串转换为有符号字节
atof()	将字符串转换为浮点数
atoi()	将字符串转换为 16 位有符号整型
atol()	将字符串转换为长整型
itoa()	将 16 位有符号整型转换为字符串

3. 存储器和字符操作

此类函数用于对存储器进行操作。其中大多数函数由 C 标准库演化而来。表 14-4

给出了一些例子。

405

表 14-4 存储器和字符串操作函数示例

函 数	功 能
memchr()	在指定存储区中查找某个值
memcmp()	比较两个数组内容
memcpy()	将一个缓冲区从数据存储器或程序存储器复制到数据存储器
memset()	使用重复的存储值初始化数组

4. 延时

表 14-5 给出了所有延时函数。其中第 1 个函数可以实现固定的单周期延时，编译时与 **nop**(无操作)指令相同。其他函数可以实现可编程的延时，与微控制器的指令周期有关。这在嵌入式环境下是非常有用的工具。

表 14-5 C18 通用软件库中的延时函数

函 数	功 能
Delay1TCY()	延时 1 个指令周期
Delay10TCYx()	延时 10 个指令周期的整数倍
Delay100TCYx()	延时 100 个指令周期的整数倍
Delay1KTCYx()	延时 1000 个指令周期的整数倍
Delay10KTCYx()	延时 10000 个指令周期的整数倍

5. 复位

程序员可以利用此类函数来检测复位原因，这些信息来自 18 系列 **RCON** 寄存器(见图 12-14)。表 14-6 给出了相关示例。

表 14-6 C18 通用软件库中的复位函数示例

函 数	功 能
isBOR()	确定复位原因是否为欠压复位
isLVD()	确定复位原因是否为低压检测复位
isMCLR()	确定复位原因是否为主清零复位
isPOR()	确定复位原因是否为上电复位

14.8.4 数学库

此类库提供了 C 标准库中所需的数学函数。变量通常为浮点型。表 14-7 给出了示例。

表 14-7 C18 数学库函数示例

函数	功能
sin()	计算正弦
cos()	计算余弦
tan()	计算正切
sqrt()	计算平方根
log10()	计算以 10 为底的对数
pow()	计算指数 x^y

14.9 深度阅读

在学习C语言的过程中，有必要利用不同类型的参考文献，用以拓宽相关的知识视野。某些文献对C语言进行了总体介绍，如参考文献 14.4。通常情况下，这类文献大多基于台式计算机的编程环境，强调从键盘输入数据并向计算机屏幕输出数据。另外还有一些有用的参考书，如参考文献 14.5。这类书籍对C语言进行了完整介绍，在结构上并没有按照教科书的形式进行组织，但对于快速查找单个内容或者了解语法细节等都非常有利。另外，还有很多参考书以嵌入式环境为目标，例如参考文献 14.6 和参考文献 14.7。这与本书的目标更为接近，但受限于具体的微处理器和编译器。因此，从某种程度上来讲，这些书中谈到的相关细节可能并不适用于其他类型的处理器。 406

小结

- ☐尽管C语言是一种高级语言，但它所具有的特性允许它在嵌入式系统中极为高效地工作。在很多嵌入式应用中，C语言是高级语言的首选。
- ☐C语言的核心特性相对简单而合理，学习起来不会太困难。
- ☐C编译器 MPLAB C18 是一款高效的软件工具，充分利用了C语言的强大特性，并针对 PIC 18 系列进行了优化。
- ☐C编译器 MPLAB C18 可以在 MPLAB IDE 环境下使用。因此，开发人员在 MPLAB IDE 环境下所积累的经验可以立即应用到C程序的开发中去。
- ☐编写C程序不仅需要了解语言自身的知识，同时也需要了解与所用编译器和处理器相关的那些库函数。C18 编译器中包含大量的库函数，包括通用功能、外围设备控制、数据操作和数学函数等。

参考文献

14.1. MPLAB C18 C Compiler Getting Started (2003). Microchip, DS51295B.
14.2. MPLAB C18 C Compiler User's Guide (2005). Microchip, DS51288J.
14.3. MPLAB C18 C Compiler Libraries (2005). Microchip, DS51297F.

407

14.4. Austin, M. and Chancogne, D. (1999). *Engineering Programming in C, Matlab and Java*. Wiley, New York. ISBN 0-471-00116-3.

14.5. Prinz, P. and Kirch-Prinz, U. (2003). *C Pocket Reference*. O'Reilly. ISBN 0-596-00436-2.

14.6. Pont, M. J. (2002). *Embedded C*. Addison-Wesley, Great Britain. ISBN 0-201-79523-X.

408

14.7. van Sickle, E. (2003). *Programming Microcontrollers in C*, 2nd edn. Elsevier, USA. ISBN 1-878707-57-4.

第 15 章
C 语言与嵌入式环境

第 14 章已经简单介绍了 C 语言。本章从现在开始将应用这些技巧来编写实际的嵌入式 C 程序。首先将介绍嵌入式条件下的重要操作，即对各个位的处理，然后将转入与外围设备的交互方法。在这个过程中，将按照例程中的出现顺序逐步介绍更多的 C 语言知识。

本章给出的大多示例都适用于 Derbot-18 AGV。这些程序都可以仿真运行，因此无论是否具有硬件都没有关系。这些程序多数都是由本书早先出现的汇编程序直接重写而来的。通过改写汇编程序，可以对不同的编程语言进行比较。阅读了本书前面内容的读者，对于程序中提到的目标硬件应该会非常熟悉。

通过本章的学习，可以很好地理解并掌握以下内容：

- ☐ 如何访问和操作单个位；
- ☐ 如何编写简单的函数并进行调用；
- ☐ 如何调用库函数，包括那些用于控制微控制器外围设备的库函数；
- ☐ 如何为 C 程序赋予某些结构，合理使用函数以及循环与分支指令。

15.1 使 C 语言适用于嵌入式环境

既然我们已经采用了高级语言(HLL)，那么就需要在两个世界里寻找最佳的解决方案。既要利用 HLL 的有利条件，同时又要保留与硬件的密切接触(如对外围设备的设置、对各个端口位的置位与清零，以及对多种中断的设置与响应)。这二者之间的兼顾与平衡是通过一些有趣的方式来解决的，下面将对此进行介绍。

15.2 位值的控制与分支

例程 14-1 以非常简单的方式说明了微控制器端口的使用。它通过下述方式来实现：在头文件中声明端口 SFR(特殊功能寄存器)，通过写 TRIS 寄存器来设置数据方向，然后通过写整个字节将数据送入端口。

对于嵌入式系统，开发程序最基本的要求是能够对单个位进行读写，这是毫无疑问的。例程 15-1 将 Derbot 上的微动开关的状态显示在 LED 上，给出了完成这一操作的 C 语言代码。事实上，它是重写例程 7-1 的结果。程序中包含有必不可少的 **main** 函

数，另外还有 2 个用户定义函数——用于初始化的 **initialise()** 和用于诊断功能的 **diagnostic()**。在程序清单中可以找到这 3 个函数。在实现 LED 的诊断闪烁功能时，程序中还用到了 C18 库函数。函数的使用方法将在 15.3 节中进行介绍。

例程 15-1 Derbot——将微动开关的状态显示在 LED 上

```
/*********************************************************************
Sw_to_led_18C                                    Uses PIC 18F242
Runs on Derbot-18. Moves state of front microswitches to leds
Files c018i.o and p18f242.lib are included by the Linker Script.
TJW 22.10.05                                     Tested 24.10.05
*********************************************************************
Clock is 4MHz
Configuration Word all default, except: crystal oscillator (HS),
power-up timer on, brown-out detect off, WDT off, LV Program disabled*/

       #include <p18F242.h>
       #include <delays.h>          //header file for delays

//Function Prototypes (Library prototypes are in Header files)
       void initialise (void);
       void diagnostic (void);

void main (void)
{
       initialise();          //call initialise function
       diagnostic();          //call diagnostic function

//move microswitch states to diag leds
loop:
       if (PORTBbits.RB4 == 0)
       PORTCbits.RC6 = 0;
       else PORTCbits.RC6 = 1;

       if (PORTBbits.RB5 == 0)
       PORTCbits.RC5 = 0;
       else PORTCbits.RC5 = 1;

       goto   loop;
}

//Initialises SFRs, and sets initial outputs.
//Assumes hardware is "Build Stage 1". All unused port bits set to output.
void initialise (void)
{      TRISA = 0b00000000; //All bits output (none used in this program)
       TRISB = 0b00110000; //Bits 5 and 4 (microswitches) only are input
       TRISC = 0b10000000; //All bits output, except bit 7 (mode switch)
//Switch all outputs off
       PORTA = 0;
       PORTB = 0;
       PORTC = 0;
}
```

```
//Diagnostic: switches leds on for 1s (Tcy = 1us)
void diagnostic (void)
{       PORTCbits.RC6 = 1;
        PORTCbits.RC5 = 1;
        Delay10KTCYx (100);
        PORTCbits.RC6 = 0;
        PORTCbits.RC5 = 0;
        Delay10KTCYx (100);
}
```

15.2.1 控制各个位

每个端口的各个位定义在微控制器头文件中，定义时采用了17.9节所述的一种C程序结构。要理解上述例程，只要知道端口位是通过 **PORTxbits. Rxy** 的格式来描述的就可以了，其中 *x* 表示端口，*y* 表示端口中的某一位。例如，在程序清单最后的诊断函数中，端口C的位5与位6通过下述代码行设置为逻辑1：

```
PORTCbits.RC6 = 1;
PORTCbits.RC5 = 1;
```

使用上述简单的步骤，只要事先声明，就可以对寄存器中的各个位进行置位或清零。由于微控制器中的所有SFR及其各个位都已在头文件中声明过，因此这将给操作带来极大的便利。

15.2.2 if与if-else条件分支结构

例程中所执行的动作是围绕 **if-else** 条件分支结构而建立的。通过它可以在程序中对两条独立的动作路径进行选择。在 **main** 函数中出现了这种结构，引用如下：

```
if (PORTBbits.RB4 == 0)
PORTCbits.RC6 = 0;
else PORTCbits.RC6 = 1;
```

上述语句可以这样解释：如果端口B的位4处于逻辑0，那么将端口C的位6设置为0；否则(其他情况)将其设置为1。与if和else相对应的语句不仅可以是单个代码行，也可以是代码块，但此时必须用花括号括起来。例如：

```
if (PORTBbits.RB4 == 0)
{PORTCbits.RC6 = 0;
PORTCbits.RC0 = 1;
}
else PORTCbits.RC6 = 1;
```

在上述程序中，当发现端口B的位4为0时，将会改变端口C中2个位的状态。 411

另外，也可以单独使用 **if** 结构。此时，当条件测试为假时，没有其他的可选动作。例如：

```
if (PORTBbits.RB4 == 0)
PORTCbits.RC6 = 0;
if (PORTBbits.RB5 == 0)
PORTCbits.RC5 = 0;
```

在此例中，当端口 B 的位 4 为 0 时，端口 C 的位 6 被置为 0，但当端口 B 的位 4 处于逻辑 1 时，并没有执行动作。同样的情况出现在后续两行代码中，对两个端口的位 5 的操作与此类似。

在这些例子中，请注意区分赋值运算符"＝"和等于运算符"＝＝"的使用方法。正如程序中所显示的那样，前者用于为变量赋值，后者则用在 if 结构内部，用于测试某个变量是否等于某个特定值。

15.2.3 设置配置位

程序清单中给出了配置位的相关设置。目前暂时可以在 MPLAB 的 **Configuration Bits** 窗口中进行设置，如图 12-15 所示。当然，这种设置方式并不非常令人满意；第 17 章将给出另一种方式，将这些设置嵌入到程序中。注意，在窗口中右击，并单击对话框中的 **Reset to Defaults** 按钮，将使窗口中的所有设置恢复到默认条件。

15.2.4 仿真并运行例程

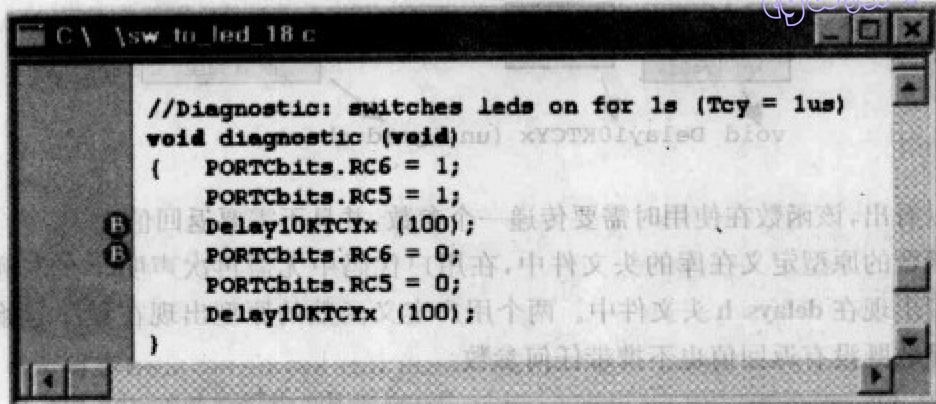
在 MPLAB® 仿真器中对这个程序进行仿真是一项很有趣的工作。首先建立项目并进行构建，然后按照下述步骤进行仿真：

- ☐ 在 MPLAB 中，单击 **Debugger ＞ Select Tool ＞ MPLAB SIM** 选择仿真器；
- ☐ 打开 Watch 窗口，选择端口 B 和端口 C 作为显示变量；
- ☐ 建立激励控制器(stimulus controller)，并选择 **Toggle** 作为 **Action**，用于模拟微动开关输入 RB5 和 RB4；
- ☐ 在 **diagnostic()** 函数中放置断点，如图 15-1a 所示；
- ☐ 使用 **Debugger ＞ Settings ＞ Osc/Trace**，将处理器频率设置为 4 MHz；
- ☐ 使用 **Debugger ＞ Stopwatch**，显示 Stopwatch 窗口，如图 15-1b 所示。

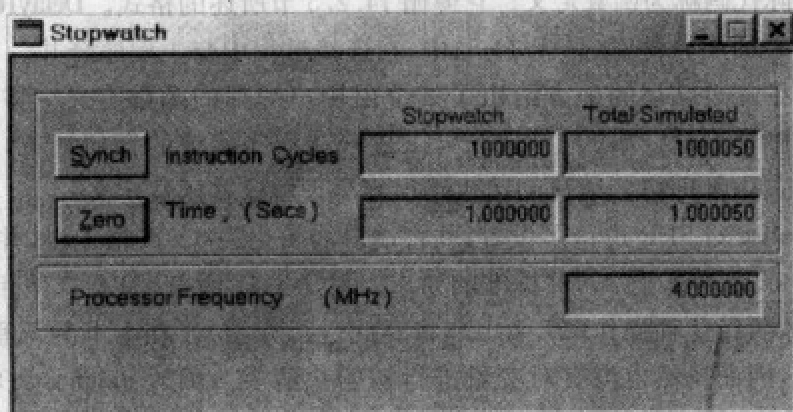
此时重启程序并运行到第一个断点处，即图 15-1a 中的首个断点。此时将跑表归零，然后继续运行到下一个断点，也就是下一行。但是，要到达下一行，程序必须执行 **Delay10KTCYx()** 函数。此时，跑表状态正如图 15-1b 所示。在该函数执行过程中，仿真时间正好走过 1s。这就很好地确认了函数的准确性。

现在，可以在标号 **loop**:之后的代码行处再设置一个断点。将程序运行到该断点，然后单步执行循环体。使用激励控制器可以改变微控制器输入(端口 B 的位 4 和位 5)的状态，从而观察程序循环的响应。

如果你拥有 Derbot-18 硬件，可以按照通常的方法下载程序。程序函数非常简单，当看到自己的第一个 C 程序在硬件上运行的时候，你将会得到很大的满足。



(a) 诊断函数中的断点示例



(b) 执行完延时函数之后的跑表状态

图 15-1 例程 15-1 的仿真参数设置

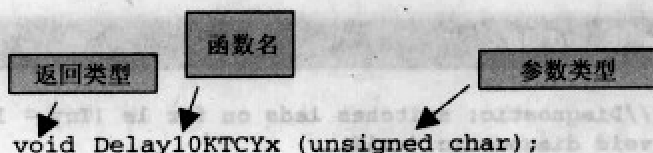
15.3 进一步认识函数

我们已经在例程中用到了一些函数，有必要进一步认识它们的使用方法。尤其需要了解以下内容：如何编写函数、如何调用函数、如何向函数传递数据以及如何从函数返回数据。

15.3.1 函数原型

由于 **main** 并不是程序中的唯一函数，因此有必要在程序中声明每个函数的函数原型（function prototype），用于通知编译器函数的参数类型（如果有的话）以及函数的返回类型。这与前面提到的函数头类似，它们的一般格式相同。例如，库函数 **Delay10TCYx( )**的原型如下：

413



可以看出，该函数在使用时需要传递一个参数，并且不需要返回值。

库函数的原型定义在库的头文件中，在用户代码中无需再次声明。在该例程中，上述原型出现在 **delays. h** 头文件中。两个用户定义函数的原型出现在程序起始处，它们表示函数既没有返回值也不携带任何参数。

15.3.2　函数定义

函数的实际代码称为*函数定义*。它遵循 14.2.5 节所述的格式。**Delay10KTCYx()** 函数的定义包含在通用软件库(见表 14-5)中，在连接过程中与主程序合并。

程序清单中靠近结尾的地方给出了 2 个由用户编写的函数定义。将其放置在这里是为了能清晰地表达。事实上，它们可以放置在程序清单的任何地方，只要不包含在其他函数定义内部即可。通过阅读代码很容易理解这些函数定义。

函数 **initialise** 对 SFR 进行了设置，并将端口初始化为 0。严格地讲，并没有必要将变量初始化为 0，因为这是 ANSI 标准的要求。对于 C18 而言，只有当使用了 c018iz. o 启动模块时，才执行这种操作(17.7.1 节将对此进行介绍)。在本书的例程中，并没有采用这种方式，因而自然也没有对变量进行初始化清零。函数 **diagnostic** 将端口 C 的位 5 和位 6(两个 LED 输出位)设置为 1，然后调用延时函数。之后，程序将这两位清零，并调用同一个延时函数。

15.3.3　函数调用与数据传递

调用函数时，需要引用函数名，并在紧随其后的括号(该括号是必需的)中放置必要的参数。如果不需要参数，仍需要保留括号，置空即可。例程 15-1 中的函数调用非常简单，也很容易理解。注意，参数 100_D 传给延时函数。

另外，还需要注意函数调用的其他重要特性，这些特性在上述特定例程中并不明显。尤其重要的是，*函数调用(function call)具有与返回类型相一致的类型与取值*。这个结论非常重要。它意味着某个调用可以插入到表达式中，此时函数将被执行并将返回值放置在表达式的对应位置上。在例程 16-1 及后续程序中，多次使用了这一方法。下面是从中摘出的例子：

414

```
ldr_rt = ReadADC()&0x03FF;  // read it, AND out unwanted bits
```

这里，语句中含有名为 **ReadADC()** 的函数调用。首先将执行该函数，然后将返回值放置在语句的对应位置上。

另外需要注意的是，传递给函数的任意参数都是原参数的一个副本，原参数值（若为已声明的变量）则被保留。这些参数副本用于函数的内部执行，最终产生返回值。在函数内部不会对原变量值进行修改。

15.3.4 延时库函数和 Delay10KTCYx( )

表 14-5 给出了 C18 通用软件库中可用的所有延时函数。它们的函数原型与前面已经见到的 **Delay10KTCYx( )**相同，并具有一个无符号字符类型参数（即 8 位的字），用作乘数以设置实际时延。

使用 **Delay10KTCYx( )**可以实现最长时延。它所引入的软件时延是 10000 个指令周期的整数倍，最大值为 255×10^4。因此，如果时钟周期为 4MHz，并将变量设置为 100（如例程所示），那么总时延就是 $10000\times100\times1$ μs，即 1s。

在例程中可以看出，程序的起始部分已经将此函数所需的头文件 **delays. h** 包含进来。

15.4 更多的分支与循环指令

15.4.1 使用 break 关键字

既然能够在 C 语言中对寄存器的单个位进行访问，那么就可以进一步采用这种方式来实现位检测与位设置的操作。

回到例程 14-2 的斐波那契程序，可以考虑用另一种方法来建立第一个循环。该循环用于生成 8 位数字（已用 **unsigned char** 加以指定）范围内的斐波那契序列。例程 14-2 以一种人为的方式来实现，将数字值限制在一个已知的"安全"上限以内。

现在，可以尝试将该循环替换为例程 15-2 中的程序段。它使用 **while(1)**结构建立了一种看似连续的循环。但是在循环内部，基于 **break** 关键字可以实现跳出。该关键字所在的语句对进位（Carry）位进行检测，当该位为高电平时跳出。如果采用这个修改后的程序进行仿真，将会看到，一旦进位位跳变至高电平，程序将在执行该语句之后立刻跳出循环。

415

例程 15-2 斐波那契程序中首个循环的另外一种写法

```
while (1)
{fibtemp = fib1 + fib2;
if (STATUSbits.C == 1) break; //exit loop if Carry bit set
 counter = counter + 1;
//now shuffle numbers held, discarding the oldest
fib0 = fib1; //first move middle number, to overwrite oldest
fib1 = fib2;
fib2 = fibtemp;
}
```

有了上述测试进位位的方法，也许有人会问：为什么不使用 **while** 条件来实现呢？此时，循环起始部分就变为：

```
while (STATUSbits.C != 1)  //loop while the Carry bit is not 1
{fibtemp = fib1 + fib2;
...
```

问题是，这种条件是在循环结束时进行检测的。此时，相加操作可能已经溢出，在斐波那契序列中至少有一个单元已经装载了不正确的数字。当然，也可以对循环中的语句进行重新组织，将加法操作放置在循环底部，这样就可以立刻对进位位进行检测。

15.4.2 使用 for 关键字

关键字 for 为循环体提供了另一种"组合(packaging)"条件的方式。它通常具有以下形式：

```
for (initialisation; condition; modification)
statement, or statements in braces
```

在这里，**for** 括号内的 3 个表达式分别称作初始值(**initialization**)、条件(**condition**)和修改(**modification**)，都由程序员进行定义。举个例子，例程 14-2 中的第一个循环可以按照下述方式重写：

```
     for (counter = 0;counter<12;counter = counter + 1)
     {fibtemp = fib1 + fib2;
//now shuffle numbers held, discarding the oldest
            fib0 = fib1;  //first move middle number, to overwrite oldest
            fib1 = fib2;
            fib2 = fibtemp;
     }
```

416

在第 1 个表达式中，**counter** 初始化为 0。当进入循环时，该值仅出现一次。需要测试的条件是 **counter** 是否小于 12，执行的修改是将 **counter** 的值加 1，这在第 1 次循环中不发生。在程序运行过程中，循环被反复执行，每次将 **counter** 的值加 1。当该值递增到 12 时，条件表达式将立刻检测到这一状态，不再执行循环。

可以对斐波那契程序进行修改，在其中添加此代码并进行仿真，从中可以得到不少乐趣。

与 **for** 相关的 3 个表达式中的任意一个都可以省略。如果将条件置空，那么将不执行测试，循环将连续执行。但是，仍然可以使用初始值和修改。创建连续循环的简单方式就是不填入任何表达式，如下所示：

```
for(;;)
{...
```

这直接等价于：

```
while(1)
{...
```

15.5 使用定时器与 PWM 外围设备

现在,我们将通过使用库函数来控制微控制器外围设备。这里需要使用 Derbot “盲目导航”程序,它最初是在例程 8-4 中介绍的。它所对应的 C 语言版本为例程 15-3。该程序使用了 Timer 2 和 PWM 所对应的库函数,并编写了一些专用函数。它们近乎是对原始程序中各个子例程的复制或重写。

程序只是简单地让 Derbot 向前运动,直至撞到障碍物(通过微动开关来检测)。然后,它将倒退并转向,如果左边的微动开关被撞则向右转向,对于右边的微动开关则执行相反的动作。在这之后,它将继续向前运动。这些简单的移动需要使用微控制器的 PWM 功能,从而需要对 Timer 2 进行设置。

例程 15-3 Derbot“盲目导航”程序

```
/**********************************************************************
Dbt_blind_Nav_PWM_C
Derbot moves by "blind" navigation.
Moves forward, and reverses and turns on bump.
Files c018i.o and p18f242.lib are included by the Linker Script.
Fixed rate PWM applied to set reasonable speeds.
TJW 3.11.05                                           Tested 7.11.05
**********************************************************************
Clock is 4MHz
Configuration Word all default, except: crystal oscillator (HS),
power-up timer on, brown-out detect off, WDT off, LV Program disabled*/

#include <p18F242.h>
#include <delays.h>         //header file for delays
#include <timers.h>         //header file for Timers
#include <pwm.h>            //header file for PWM

/*function prototypes, reproduced from Header Files for information
      void OpenPWM1 (char);
      void OpenPWM2 (char);
      void OpenTimer2 (unsigned char);
      void Delay10KTCYx (unsigned char); */

//User-defined function prototypes
      void diagnostic (void);
      void leftmot_fwd (void);
      void rtmot_fwd (void);
      void rev_left (void);
      void rev_rt (void);

void main (void)
{
/*Initialises SFRs, and sets initial outputs. Assumes hardware is "Build
Stage 2". All unused port bits set to output. Used bits are identified.*/
```

```
    TRISA = 0b00000000;  //All bits output, 2 & 5 used for motor enables.
    TRISB = 0b00110000;  //Bits 5 and 4 (microswitches) only are input,
    TRISC = 0b10000000;  //All bits output except 7 (mode switch),
                         //1 & 2 used for PWM
    ADCON1 = 0b00000110; //Set Port A for digital i/o

//Switch all outputs off
    PORTA = 0;
    PORTB = 0;
    PORTC = 0;
//call diagnostic function
    diagnostic();
//Enable PWM
    OpenTimer2 (TIMER_INT_OFF & T2_PS_1_1 & T2_POST_1_1);
    OpenPWM1 (0xFF); //Enable PWM1 and set period
    OpenPWM2 (0xFF); //Enable PWM2 and set period

    while (1)
    {
    //start motors
    leftmot_fwd ();
    rtmot_fwd ();

//test for bumps - reverse and turn if either microswitch closes
    if (PORTBbits.RB4 == 0) //Test right uswitch
    rev_left ();
    if (PORTBbits.RB5 == 0) //Test left uswitch
    rev_rt ();
    Delay10KTCYx (10);
    }
}
/***************************************************************
Motor Drive Functions
***************************************************************/
    void leftmot_fwd (void)     //sets left motor running forward
{   CCPR2L = 196;
    PORTAbits.RA5 = 1;          //enable motor
}

    void rtmot_fwd (void)       //sets right motor running forward
{   CCPR1L = 196;
    PORTAbits.RA2 = 1;          //enable motor
}

    void leftmot_rev (void)     //sets left motor running in reverse
{   CCPR2L = 60;
    PORTAbits.RA5 = 1;          //enable motor
}

    void rtmot_rev (void)       //sets right motor running in reverse
{   CCPR1L = 60;
```

418

```
    PORTAbits.RA2 = 1;          //enable motor
}

    void rev_rt (void)          //reverses and then turns to right
{   PORTCbits.RC6 = 1;          //set right led
    PORTAbits.RA5 = 0;          //stop motors
    PORTAbits.RA2 = 0;
    PORTBbits.RB1 = 1;          //small bleep from sounder
    Delay10KTCYx (50);
    PORTBbits.RB1 = 0;          //clear sounder
    leftmot_rev ();             //reverse both motors
    rtmot_rev ();
    Delay10KTCYx (200);
    leftmot_fwd ();             //left motor forward to turn
    Delay10KTCYx (100);
    PORTCbits.RC6 = 0;          //clear led
}

    void rev_left (void)        //reverses and then turns to left
{   PORTCbits.RC5 = 1;          //set left led
    PORTAbits.RA5 = 0;          //stop motors
    PORTAbits.RA2 = 0;
    PORTBbits.RB1 = 1;          //small bleep from sounder
    Delay10KTCYx (50);
    PORTBbits.RB1 = 0;
    leftmot_rev ();             //reverse both motors
    rtmot_rev ();
    Delay10KTCYx (200);
    rtmot_fwd ();               //right motor forward to turn
    Delay10KTCYx (100);
    PORTCbits.RC5 = 0;          //clear led
}
...
diagnostic function same as Program Example 15.1.
...
```

15.5.1 使用定时器外围设备

18FXX2 有 4 个定时器，每个定时器都有 4 个库函数。它们显示在表 15-1 中，其中 x 可以是 0、1、2、3。参考文献 14.3 给出了相关参数的完整内容。

例程中仅使用了 1 个定时器相关函数 **OpenTimer2()**。它代表了一种外围设备驱动库函数类型，下面将再次看到。其中，参数由位屏蔽构成，通过在多个设置上施加逻辑"与"来实现。在库参考文献 14.3 中对这些设置进行了描述，表 15-2 给出了该函数所对应的各种设置。在该例中，需要选择 3 种设置，包括中断启用、预分频与后分频。在函数调用中可以看到它们，引用如下：

```
OpenTimer2 (TIMER_INT_OFF & T2_PS_1_1 & T2_POST_1_1);
```

表 15-1 定时器库函数

函 数	动 作
OpenTimerx()	配置 Timerx
ReadTimerx()	读取 Timerx
WriteTimerx()	写入 Timerx
CloseTimerx()	关闭 Timerx

表 15-2 OpenTimer2()的设置选项

取 值	作 用
中断	
TIMER_INT_ON	中断启用
TIMER_INT_OFF	中断禁止
预分频比	
T2_PS_1_1	1:1 预分频
T2_PS_1_4	1:4 预分频
T2_PS_1_16	1:16 预分频
后分频比	
T2_POST_1_1	1:1 后分频
T2_POST_1_2	1:2 后分频
…	…
T2_POST_1_16	1:16 后分频

上述代码启用定时器，禁止中断，并将预分频比和后分频比均设置为 1:1。事实上，它用于取代下列汇编代码（引自例程 9-2）：

```
movlw B'00000100'   ;switch on Timer2, no pre or postscale
movwf t2con
```

在减少代码行方面，C 语言的优势并不明显，它的优势体现在其他方面。使用上述库函数，程序员就无需深入了解外围设备结构以及相关的 SFR 位的细节。只要理解函数规范，就可以使用外围设备，所需了解的内部工作机理是很少的。

15.5.2 使用 PWM

在 9.5 节中，已经介绍了 PWM 的概念以及外围设备的使用方法。其硬件围绕 Timer 2 而建立，刚刚接触时也许较难理解。但是，使用起来却并不困难。表 15-3 给出了适用于 18FXX2 微控制器的 PWM 库函数，其中 x 可以取值 1 或 2。

例程 15-3 在下列两行代码中使用了函数 **OpenPWMx()**：

```
OpenPWM1 (0xFF);  //Enable PWM1 and set period
OpenPWM2 (0xFF);  //Enable PWM2 and set period
```

该函数启用 CCP 模块的 PWM 模式，并装载 **PR2** 寄存器（参见图 9-11）。当然，**PR2** 寄存器由各个 CCP 模块共用，只能设置为同一个值。因此，要求两个函数调用中的参数相同。要设置循环速率，需要使用等式(9-2)。此时，Timer 2 不包含预分频，函

数参数取 $0xFF_H$，从而得到 PWM 频率为 3.906kHz。

在上述例程中，并没有使用 **SetDCPWMx()** 函数来设置或改变速率。由于仅使用 8 位精度，因此可以对 **CCPR1L** 和 **CCPR2L** 寄存器直接写入。

表 15-3 PWM 库函数

函 数	动 作
OpenPWMx()	配置 PWM x 的周期和时基
SetDCPWMx()	向 PWM x 写入 10 位占空比取值
ClosePWMx()	禁止 PWM x

421

15.5.3 主程序循环

例程仍然使用 **while(1)** 结构来构成主程序循环，并使用 **if** 语句对微动开关逐个进行检测，如下所示：

```
if (PORTBbits.RB4 == 0)   //Test right uswitch
rev_left ();

if (PORTBbits.RB5 == 0)   //Test left uswitch
rev_rt ();
```

当任何微动开关被触动时，与之对应的输入值变为 0。然后程序调用 **rev_left** 或 **rev_rt**。这些函数理解起来并不困难。此时，AGV 将停止运动，两个电机倒转一个固定的时间。然后，其中的某个电机开始正转(另一个则依然倒转)，促使 AGV 转向。然后，程序执行将返回主循环，AGV 再次向前运动。该循环使用了电机驱动函数和 **Delay10KTCYx()** 函数。

小结

本章从实践的角度讲解了如何在嵌入式环境和 PIC® 18 系列微控制器中使用 C 语言。

- ☐ 存储寄存器中的各个位可以方便地进行访问与操作。
- ☐ 有很多分支与循环结构可以用来清晰地定义程序流程。
- ☐ 可以方便地区分并使用各种库函数，它们极大地简化了与微控制器外围设备之间的交互。
- ☐ 函数的编写与使用并不困难；在不同的函数中定义不同的任务，可以得到好的程序结构；在主程序中可以包含大量的函数调用。

422

第 16 章
使用 C 语言实现数据的采集与使用

前面已经建立了嵌入式 C 语言编程环境的基础，现在将进一步认识如何用 C 语言来实现数据的采集与处理。

首先，我们感兴趣的是如何使用 C 语言与 ADC 外围设备进行交互。有很多有用的库函数可以辅助完成这项任务。在数据采集完成之后，需要考虑如何进行后续的存储和处理。这些将引领我们认识 C 数组与字符串。由于需要将数据移动到其他地方，因此需要了解 I^2C 外围设备的使用方法。总的来说，这可能涉及复杂的学习领域，但我们只是介绍一些入门级的知识，仅考虑整数数据的处理。

与前几章类似，本章所给的例程大多应用于 Derbot-18 AGV，但是它们都可以通过仿真来获得良好的效果。

学习本章内容，你将会理解以下内容：

- ☐ 如何调用库函数使用 18FXX2 ADC 和 I^2C 外围设备；
- ☐ 如何处理数组、字符串和指针；
- ☐ 如何调用与字符串处理相关的库函数。

16.1 用 C 语言实现数据处理

强大的数据处理能力是 C 语言的优势之一。它可以定义数据类型，控制数据移动，并保护数据避免不期望的更改。在台式电脑中，C 语言提供有很多库函数，可以方便地实现数据块的移动。在本章中，将会看到一些类似的能力如何应用到嵌入式环境中。虽然前面已经提到，本章仅使用整数是出于简化的目的，但仍值得一提的是，浮点程序在 8 位微控制器(如 18 系列)上执行是相当耗时的。因此，除非绝对必要，应当尽量避免使用浮点运算。

16.2 使用 18FXX2 ADC

例程 16-1 提供了一个内容更加丰富的 C 语言编程示例，同样适用于 Derbot-18 AGV。这是一个寻光程序，最先在例程 11-3 中以汇编语言引入。Derbot 基于 3 个光敏电阻器(LDR)对光源进行搜索，当所有传感器感受到的光强相近似时，Derbot 就停

止运动。程序提供了更多条件分支的有用示例,并介绍了ADC的使用方法。注意,在程序注释中嵌入了一些代码行编号。

例程16-1 Derbot"寻光"程序

```
/*********************************************************************************
Dbt_light_seek_c
Derbot seeks light. PWM applied. Speed is dependent on light difference
(front to back), so Derbot comes to a halt when light difference is minimal.
Microswitches used for bump detection.
Files c018i.o and p18f242.lib are included by the Linker Script.
TJW 12.11.05                                                  Tested 27.11.05
*********************************************************************************
Clock is 4MHz.
Configuration Word all default, except: crystal oscillator (HS),
power-up timer on, brown-out detect off, WDT off, LV Program disabled*/

#include <p18F242.h>
#include <adc.h>
#include <timers.h>
#include <pwm.h>
#include <delays.h>

/*function prototypes, reproduced from Header Files for information only
      void OpenPWM1 (char);
      void OpenPWM2 (char);
      void OpenTimer2 (unsigned char);
      void Delay10KTCYx (unsigned char);
      void Delay10TCYx (unsigned char);
      void OpenADC (unsigned char, unsigned char);
      void SetChanADC (unsigned char);
      void ConvertADC(void);
      char BusyADC(void);
      int ReadADC(void);                      */

//User-defined function prototypes
      void leftmot_fwd (void);
      void rtmot_fwd (void);
      void leftmot_rev (void);
      void rtmot_rev (void);
      void rev_left (void);
      void rev_rt (void);
      void fwd_left (void);
      void fwd_rt (void);
      void rotate_rt (void);
      void rotate_left (void);
      void diagnostic (void);

//Declare Variables
      int ldr_rt;          //right ldr value
      int ldr_left;        //left ldr value
      int ldr_rear;        //rear ldr value
      int ldr_ave;         //computed average of front ldrs
      int ldr_diff;        //difference between front ldrs, left - right
      int ldr_fwd;         //ave fwd speed required
```

```
  int fwd_dr_left;     //offset added to left PWM for fwd motion
  int fwd_dr_rt;       //offset added to right PWM for fwd motion
//Main Program
      void main (void)
{
//Line 57 Initialise. Active bits identified. Unused bits set as outputs.
      TRISA = 0b00001011;  //ADC channels set as inputs,
                                        //bits 2&5 are motor enables
      TRISB = 0b00110000;  //bits 4 & 5 are uswitch inputs
      TRISC = 0b10000000;  //bit 7 is mode switch, 1 & 2 are PWM
//Enable Timer 2, with pre- and post-scalers divide-by-1
      OpenTimer2  (TIMER_INT_OFF & T2_PS_1_1 & T2_POST_1_1);
      OpenPWM1 (0xFF);     //Enable PWM1 and set period
      OpenPWM2 (0xFF);     //Enable PWM2 and set period
//Enable ADC. Port A bits 0,1,3 are analog input, internal reference,
      //right justify result
OpenADC(ADC_FOSC_8 & ADC_RIGHT_JUST & ADC_3ANA_0REF,ADC_CH0 & ADC_INT_OFF)
//Switch all outputs off
      PORTA = PORTB = PORTC = 0;
//call diagnostic function
      diagnostic();
//enable motors at idle speed
      CCPR1L = CCPR2L = 0x80;
      PORTAbits.RA5 = 1;
      PORTAbits.RA2 = 1;

//**************************************************************************
//this is main loop.
//**************************************************************************
while (1)
  {
//Line 83 check first for collisions
      if (PORTBbits.RB4 == 0) //Test right uswitch
      rev_left ();
      if (PORTBbits.RB5 == 0) //Test left uswitch
      rev_rt ();
//Read and store all ldr values
      //left channel
      SetChanADC (ADC_CH0);
      Delay10TCYx (2);              //delay for 20us approx acquisition time
      ConvertADC();
      while (BusyADC());            //wait for conversion to complete
      ldr_left = ReadADC()&0x03FF;  // read it, AND out unwanted bits
      ldr_left = 1024 - ldr_left;   //reverse polarity
      //Line 96  right channel
      SetChanADC (ADC_CH1);
      Delay10KTCYx (2);             //delay for 20us approx acquisition time
      ConvertADC();
      while (BusyADC());
      ldr_rt = ReadADC()&0x03FF;  // read it, AND out unwanted bits
      ldr_rt = 1024 - ldr_rt;     //reverse polarity
      //rear channel
      SetChanADC (ADC_CH3);
```

```
    Delay10KTCYx (2);           //delay for 20us approx acquisition time
    ConvertADC();
   while (BusyADC());
   ldr_rear = ReadADC()&0x03FF;  // read it, AND out unwanted bits
   ldr_rear = 1024 - ldr_rear;   //reverse polarity
//Line 110 Compute some intermediate variables
   ldr_diff = (ldr_left - ldr_rt);  //difference between ldrs
   ldr_ave = (ldr_left + ldr_rt);   //average front two ldrs
   ldr_ave = (ldr_ave>>1);          //divide this +ve no. by 2
   ldr_fwd = ldr_ave - ldr_rear;  //fr. to back difference, for fwd speed
   if (ldr_fwd < 0) ldr_fwd = 0;   //set minimum value
//Line 116 determine action, by comparing LDR readings
   if (ldr_left > ldr_rt)
   {if (ldr_left > ldr_rear)
        fwd_left();             //ldr_left is brightest, go forward left
    else rotate_left ();        //rear is brightest, rotate towards light
   }
   else
   {if (ldr_rt > ldr_rear)
        fwd_rt();               //ldr_rt is brightest, go forward left
        else rotate_rt ();
   }
   Delay10KTCYx (10);
  }          //end of while
 }           //end of main

/*****************************************************************
Movement Functions
One of these four functions selected every loop iteration
*****************************************************************/
/*light is front left, hence move forward left. Algorithm is:
fwd drive left = ldr_fwd - ldr_diff, fwd drive right = ldr_diff + ldr_fwd*/
   void fwd_left (void)
   {
   fwd_dr_left = ldr_fwd - ldr_diff;
   if (fwd_dr_left < 0) fwd_dr_left = 0;  //set to zero if -ve
   fwd_dr_left = fwd_dr_left>>1;    //rotate right to scale down value
   if (fwd_dr_left > 127) fwd_dr_left = 127;  //limit maximum value
   fwd_dr_rt = ldr_fwd + ldr_diff;
   if (fwd_dr_rt < 0) fwd_dr_rt = 0;        //set to zero if -ve
   fwd_dr_rt = fwd_dr_rt>>1;        //rotate right to scale down value
   if (fwd_dr_rt >127) fwd_dr_rt = 127;    //limit maximum value
   CCPR1L = 0x80 + fwd_dr_rt;        //set right motor, which is greater
   CCPR2L = 0x80 + fwd_dr_left;  //set left motor, which is less
   }
/*light is front right, hence move forward right. Algorithm is:
fwd drive right = ldr_fwd + ldr_diff, fwd drive left = ldr_fwd - ldr_diff
(noting "polarity" of ldr_diff*/
   void fwd_rt (void)
   {
   fwd_dr_rt = ldr_fwd + ldr_diff;
   if (fwd_dr_rt < 0) fwd_dr_rt = 0;        //set to zero if -ve
   fwd_dr_rt = fwd_dr_rt>>1;        //rotate right to scale down value
```

426

```
        if (fwd_dr_rt > 127) fwd_dr_rt = 127;      //limit maximum value
        fwd_dr_left = ldr_fwd - ldr_diff;
        if (fwd_dr_left < 0) fwd_dr_left = 0;      //set to zero if -ve
        fwd_dr_left = fwd_dr_left>>1;          //rotate right to scale down value
        if (fwd_dr_left >127) fwd_dr_left = 127;  //limit maximum value
        CCPR1L = 0x80 + fwd_dr_rt;           //set right motor, which is less
        CCPR2L = 0x80 + fwd_dr_left;         //set left motor, which is greater
        }

//fixed speed left rotation (light is at rear left)
        void rotate_left (void)
        {rtmot_fwd ();
        leftmot_rev ();
        }

//fixed speed right rotation (light is at rear right)
        void rotate_rt (void)
        {leftmot_fwd ();
        rtmot_rev ();
        }
/***************************************************************
Motor Drive Functions
***************************************************************/
...
(same functions as Program Example 15.3 - blind navigation program)
...
```

上述例程比较复杂,可以首先观察一下该程序的结构。其中采用了很多编程习惯,都是为了提高程序的清晰度。这里用到的编程习惯是:在存在嵌套代码块的地方,每个嵌套代码块的起始括号都向右缩进一步。首先,**main** 函数的起始括号位于最左侧。然后,请注意第 83 行出现的 **while** 主循环的起始括号的位置,它向右缩进一个制表符。循环中包含的其他代码块则进一步向右缩进,相互匹配的括号总是在垂直方向上相互呼应。**while** 循环和 **main** 的结尾处都添加了注释,这样就可以方便地看到它们。在本书中,位于“最左侧”的括号对通常留给 **main** 函数。因此,相对较小的函数括号需要缩进,但仍需在垂直方向上成对出现。主要注释列写在单行或多行中,而较短的注释则放置在代码右侧。

16.2.1 寻光程序的结构

该程序大约包含 20 个函数,演示了函数个数是如何随着程序复杂性的提高而增多的。其中大约一半函数来自已有的库,其他则由用户所定义。库函数的原型出现在相关头文件中,因而无需在源文件中加以说明。但在这里,以注释的形式给出,用于提供相关信息。其他函数的原型则真实地包含在源文件中。

与通常的做法类似,**main** 函数以初始化开头,紧接着是诊断函数。然后,程序使用 427 **while** 关键字进入连续循环。在循环中,程序首先检测前侧微动开关,必要时作出响应,如例程 15-3 所示。然后程序大致遵循图 11-13 所示的流程图执行。

16.2.2 使用 ADC

11.3 节描述了 16F873A 和 18F242 中的 ADC 结构。表 14-1 显示了与 ADC 相关的所有 6 个函数。例程使用了其中的 5 个。其中最复杂的函数是 **OpenADC**，在程序起始部分已经引用了该函数的原型，这里再次写出：

```
void OpenADC (unsigned char, unsigned char);
```

其中用作参数的两个无符号字符由多个位屏蔽组成，使用方法与 15.5.1 节中的 **OpenTimer2()** 函数类似。ADC 的选项比 Timer 2 更加复杂。这里没有对它进行相关描述，但可以在参考文献 14.3 中找到。这里用到的函数如下：

```
//Enable ADC. Port A bits 0,1,3 are analog input, internal reference,
        //right justify result
OpenADC(ADC_FOSC_8 & ADC_RIGHT_JUST & ADC_3ANA_0REF,ADC_CH0 & ADC_INT_OFF);
```

正如注释所述，该函数可以实现以下设置。

- ☐ 设置 ADC 的转换速率，具体值取决于驱动 ADC 的振荡器。最小转换时钟周期(T_{AD})为 1.6μs，对 4MHz 的内部振荡器进行 8 分频可以将 T_{AD} 设置为 2.0μs。
- ☐ 结果为右对齐，则如图 11-9 所示。
- ☐ 使用 3 个通道作为输入，采用内部参考电压。将寄存器 **ADCON1** 的低 4 位设置为 0100，如图 11-8 所示。
- ☐ 当前选择通道 0。这不影响后续程序。
- ☐ 关闭中断。

该函数有效地代替了下列汇编代码行：

```
bsf     status,rp0
...
movlw   B'10000100'   ;select port A bits 0,1,3 for analog input
movwf   adcon1             ;right justify result
bcf     status,rp0
movlw   B'01000001'    ;set up ADC: clock Fosc/8, switch ADC on but not
                                                    ;converting,
movwf   adcon0         ;input channel selection currently irrelevant
```

后续的转换过程遵循图 11-5 所示的数据采集流程图。对每个 LDR 重复同一过程，通道 0 的情况如下所示：

```
SetChanADC (ADC_CH0);
Delay10TCYx (2);               //delay for 20us approx acquisition time

ConvertADC();
while (BusyADC());             //wait for conversion to complete
ldr_left = ReadADC()&0x03FF;  // read it, AND out unwanted bits
ldr_left = 1024 - ldr_left;   //reverse polarity
```

这里所用的函数也相当明确：**SetchanADC()** 用于选择输入通道，然后延时 20μs 以确保采集时间。使用 **ConvertADC()** 来启动转换。函数 **BusyADC()** 用于测试转换是否

正在进行，如果是则返回 1。15.3.3 节已经提到，使用函数调用等同于使用其返回值。这里将函数调用嵌入在 **while** 结构中。这样将使程序在转换结束之前始终在该点循环。

然后，通过 **ReadADC()** 函数读取转换结果。函数嵌入在表达式中，实际使用的是它的返回值，也即 ADC 转换结果。这是一个 10 位数，最大可能值为 1023_D。将其结果与 $03FF_H$ 进行"与"操作，从而确保读取时更高位为 0。虽然这步也许并不必要，但它却说明函数可以放置在表达式中。基于 LDR 的硬件结构，转换结果的数值大小随光强的增加而减小。为简化后续计算，将该结果减去 10 位最大值 1024_D，这样当光强增加时，**ldr_left** 的值就随之增大。

16.2.3 if-else 的更多应用

在数据转换之后将执行一些中间计算，然后从第 116 行开始，程序将确定（从 4 条可能路径中）选择哪条路径进行执行。这一过程遵循图 11-13 所示的流程图。当某个前侧 LDR 最亮时，它将转向该方向。当后侧 LDR 最亮时，它将顺时针或逆时针旋转。这些决策过程由下列 3 个 **if-else** 测试来实现。可以看出，首个 **if** 中嵌套了 **if-else** 结构，后面的 **else** 也是如此。

```
//determine action, by comparing LDR readings
        if (ldr_left > ldr_rt)
        {if (ldr_left > ldr_rear)
             fwd_left();             //ldr_left is brightest, go forward left
          else rotate_left ();          //rear is brightest, rotate towards light
        }
        else
        {if (ldr_rt > ldr_rear)
             fwd_rt();               //ldr_rt is brightest, go forward right
             else rotate_rt ();
        }
```

16.2.4 寻光程序的仿真

无论是否有 Derbot，都可以使用 MPLAB® 仿真器来仿真上述例程。通过仿真可以检查所用分支与操作是否正确。按照下述步骤对仿真过程进行设置。

429

- ☐ 在 MPLAB 中使用 **Debugger > Select Tool > MPLAB SIM** 选择仿真器。
- ☐ 打开 Watch 窗口，并选择图 16-1 中所示变量。
- ☐ 配置激励控制器（Stimulus Controller），并仿真微动开关的输入 RB5 和 RB4。
- ☐ 使用 **Debugger > Settings > Osc/Trace** 将处理器频率设置为 4 MHz。该设置在这里非常重要，因为如果频率设置过高，仿真器会很快识别出来，并提醒你转换周期 T_{AD}（11.3.2 节）太短。
- ☐ 在第 84 行和第 111 行插入断点。

运行程序至第 1 个断点。使用激励控制器，将端口 B 的位 4 和位 5 设置为 1，模拟

微控制器的未触动状态。现在再次运行程序到第111行的断点。输出窗口将给出警告“No stimulus file attached to ADRESL for A/D(未向A/D的ADRESL配置激励文件)”。出现该警告不必担心。

Watch

Add SFR | ADCON0 | Add Symbol | _tmp_0

Address	Symbol Name	Hex	Decimal	Binary
0082	ldr_left	0028	40	00000000 00101000
0080	ldr_rt	0050	80	00000000 01010000
0084	ldr_rear	0064	100	00000000 01100100
008A	ldr_fwd	0000	0	00000000 00000000
0088	ldr_diff	FFD8	65496	11111111 11011000
0086	ldr_ave	003C	60	00000000 00111100
008C	fwd_dr_left	0000	0	00000000 00000000
008E	fwd_dr_rt	007F	127	00000000 01111111
0FBE	CCPR1L	50	80	01010000
0FBB	CCPR2L	B0	176	10110000
0F81	PORTB	30	48	00110000

图16-1 用于“寻光”程序仿真的观察窗口

现在，从表16-1中选择实验值 **ldr_left**、**ldr_right** 和 **ldr_rear** 的第1组设置输入到观察窗口中(为每个变量输入十进制数值，其他列中的值将会随之变化)。从这里开始，使用 Step Into 调试器按钮单步执行程序。对于第1种情况，可以观察到程序是如何正确选择函数 **fwd_left** 的，并且可以在观察窗口查看计算结果。表中给出了输入变量的每组设置所对应的结果。当到达第1组结果时，运行程序回到第1个断点处，然后继续运行到第2个断点处，输入另一组结果。按照这种方式继续循环，输入不同的实验结果，观察程序在不同条件下的计算结果与循环分支。

430

表16-1 仿真“寻光”程序时的实验值

条 件	来自ADC的输入值(十进制)			最终动作与取值(十进制)		
	ldr_left	ldr_rt	ldr_rear	所选函数	fwd_dr_left	fwd_dr_rt
左>右>后	0100	0080	0040	fwd_left	015	035
右>左>后	0080	0100	0040	fwd_rt	035	015
左>>右>后	0200	0040	0020	fwd_left	00	127
后>左>右	0080	0040	0100	rotate_left	—	—
后>右>左	0040	0080	0100	rotate_right	—	—

如果你有 Derbot AGV，运行该程序将获得很多乐趣。本章后面将加入显示功能，运行效果也会变得更加有趣。

16.3 指针、数组与字符串

第 14 章已经对数据元素的声明与使用方法进行了介绍。但是，有很多数据是以变量集合的形式存在的，例如准备发送给显示设备的数据串。因此，本节将关注于数据集的定义和使用，介绍数组、字符串以及用于访问这两类数据集的指针。

16.3.1 指针

在描述变量时可以不使用名称，而使用它的地址。在 C 语言的术语中，此类地址称为指针。使用一元运算符“&”可以为指针装载变量地址，如下所示：

```
my_pointer = &fred;
```

上述代码将变量 **fred** 的地址装载在变量 **my_pointer** 中，此时称 **my_pointer** 指向 **fred**。

另一方面，如果在指针前面加上“ * ”运算符，则表示该指针所指的变量值。例如，* **my_pointer** 可以读作“由 **my_pointer** 所指向的数值”。以上述方式使用的 * 运算符有时也称作解除引用(dereferencing)运算符或间接(indirection)运算符。指针的间接值(如 * **my_pointer**)可以像任何其他变量一样在表达式中使用。

指针声明为它所指向的数据类型。如下所示：

```
int *my_pointer;
```

上式表示指针 **my_pointer** 指向一个 **int** 类型的变量。

431

由于 PIC® 微控制器的存储器为哈佛结构，因此 C18 编译器允许为程序存储器和数据存储器设置指针。表 16-2 显示了相应的指针长度。表中“近”和“远”的含义将在 17.6.5 节中讲述。

表 16-2 C18 指针长度

指针类型	指针长度
数据存储器	16 位
近程序存储器	16 位
远程序存储器	24 位

16.3.2 数组

在 C 语言中，数组定义为由多个相同类型的数据元素构成的集合。可以使用任何数据类型。数组元素存储在连续的存储单元中。定义时，需要声明数组名称、数据类型和数组中的元素个数(此为可选)。例如：

```
unsigned char message1[8];
```

该声明定义了一个名为 **message1** 的数组，包含 8 个字符。数组在名称之后使用方括号，因此可以很容易地识别出来。

数组中的元素可以通过索引进行访问，从 0 开始编号。因此，对于上述数组，**message[0]**选择首个元素，**message[7]**选择最后一个元素。该索引可以使用任何变量来替换，变量值为所需取值。

需要特别注意的是：数组名等价于首个元素的地址。因此，如果将数组名传递给函数，实际传递的是首个元素的地址。

16.3.3 对数组使用指针

使用前述运算符可以将指针指向数组。例如：

```
ADC_val_ptr = &ADC_val_BCD[0];
```

该语句将数组 **ADC_val_BCD** 的首个元素的地址分配给指针 **ADC_val_ptr**。必要时，可以将指针的赋值与声明组合在一起，例如：

```
int *ADC_val_ptr = &ADC_val_BCD[0];
```

在上述赋值语句之后，**ADC_val_BCD[0]**（数组中的首个元素）的取值就等价于 ***ADC_val_ptr**（**ADC_val_ptr** 所指向的值）。相应地，**ADC_val_BCD[1]**的取值与*（**ADC_val_ptr+1**）等价，**ADC_val_BCD[2]**的取值与*（**ADC_val_ptr+2**）等价，等等。

由于数组名等价于首个元素地址，上述赋值语句也可以写作：

```
ADC_val_ptr = ADC_val_BCD;
```

从而，**ADC_val_BCD[i]**就等价于*（**ADC_val_BCD+i**），此时事实上是将数组名用作指针。

另外，值得一提的是指针也可以与索引一起使用。因此，**ADC_val_ptr[i]**就等价于*（**ADC_val_ptr+i**）。看起来指针与数组名几乎是可以互换的，那么指针似乎并无必要。但是请注意：数组名是常量（数组是固定在内存中的），而指针则是变量，它具有变量的所有属性。

16.3.4 字符串

字符串是一种特殊类型的数组，由 **char** 类型的字符组成，并以空字符"\0"作为结尾。因此，字符串数组的长度至少应比原字符串多出 1 字节，以容纳该终止符。

16.3.5 指针、数组和字符串的应用例程

例程 16-2 描述了数组、字符串和指针的使用方法。该例程有些不太真实，它与嵌入式系统无关，仅用于仿真。程序中声明了一个名为 **list** 的字符数组，每个元素值都预设为 0。另外还声明了一个名为 **item1** 的字符串，用于容纳 Apple。类似地，还声明了

432

字符串 Pear。接着又声明了单个字符 **plural**。注意，声明时需要将字符串放置在双引号中，单个字符则放置在单引号中。最后定义了两个指针，将 **pntr1** 赋值为字符串 **item1** 首个元素的地址，将 **pntr2** 赋值为变量 **number** 的地址。

例程 16-2 处理指针、数组和字符串

```
/*****************************************************************************
Strings&chars_c
This program, for simulation only, explores characters, strings and pointers.
Files c018i.o and p18f242.lib are included by the Linker Script.
TJW 29.11.05                                                  Tested 30.11.05
*****************************************************************************/
//Configuration bits need not be set
#include <p18F242.h>

//data type definitions
      char counter;                    //index for list
      char list[8] = {0,0,0,0,0,0,0,0};
      char number = 0;
      char item1[] = "Apple";
      char item2[] = "Pear";
      char plural = 's';
      char *pntr1 = &item1[0];  //Pointer to "Apple" string
      char *pntr2 = &number;
// Main function
      void main (void)
{
loop:
//Do apples
      counter = 0;                  //set index to list
      list[counter] = number;       //indicate number of items
      while (item1[counter] != 0) //indicate type of item
      {list[counter+1] = *(pntr1+counter);
      counter = counter + 1;
      }
      if (number >1)list[counter+1] = plural;  //set the item to plural
      else list[counter+1] = 0x20;  //return item to single, with ASCII space

//Do pears
      counter = 0;                 //set index to list
      list[counter] = number;   //indicate number of items
      while (item2[counter] != 0)  //indicate type of item
      {list[counter+1] = *(item2+counter);
      counter = counter + 1;
      }
      if (number >1)list[counter+1] = plural;  //set the item to plural
      else list[counter+1] = 0x20;  //return item to single, with ASCII space
      counter = counter + 1;
      list[counter+1] = 0x20;       //insert ASCII space
      number = *pntr2 + 1;
      if (number > 9)number = 0;
      goto loop;
```

433

16.3.6 对 while 条件的补充说明

在14节中，已经对 while 关键字进行了介绍。虽然以前已经多次使用 **while** 条件实现无限循环，但是从条件判断的意义上来还是第一次使用，如下代码行：

```
while (item1[counter] != 0)   //indicate type of item
```

在接下来的几行之后又使用了类似的条件。这个条件的含义非常明确，也即由 **counter** 值所确定的数组元素必须不等于0。事实上可以将上述语句进行简化，这在第14章中已经进行了解释。只要条件表达式为“真”(即非零)，循环就继续执行，当表达式为0时程序将离开循环。因此，可以采用下面更简单的形式来编写 **while** 语句：

```
while (item1[counter])  //indicate type of item
```

上述语句与原始代码所产生的效果相同。

434

16.3.7 仿真例程

如果已安装了C18编译器，就可以围绕例程16-2(源代码在本书附属资源中)创建项目，构建之后就可以使用MPLAB仿真器进行仿真。

打开观察窗口，会显示如图16-2所示变量。通过该窗口可以查看仿真器是如何处理数组的相关显示的。从图中可以看出，可以仅使用名称来显示变量(如 **item2**)，也可以将其扩展为完整列表(如 **list** 和 **item1**)。注意 **item1** 在创建时包含6个元素，最后的元素为空字符。

Watch

Add SFR | ADCON0 | Add Symbol | item2

Address	Symbol Name	Value	Hex	Decimal	Binary
0080	⊟ list	"□Apples"			
0080	[0]	.	02	2	00000010
0081	[1]	A	41	65	01000001
0082	[2]	p	70	112	01110000
0083	[3]	p	70	112	01110000
0084	[4]	l	6C	108	01101100
0085	[5]	e	65	101	01100101
0086	[6]	s	73	115	01110011
0087	[7]	.	00	0	00000000
00A3	counter	0x00	00	0	00000000
0088	number	0x02	02	2	00000010
0089	⊟ item1	"Apple"			
0089	[0]	A	41	65	01000001
008A	[1]	p	70	112	01110000
008B	[2]	p	70	112	01110000
008C	[3]	l	6C	108	01101100
008D	[4]	e	65	101	01100101
008E	[5]	.	00	0	00000000
008F	⊞ item2	"Pear"			

Watch 1 | Watch 2 | Watch 3 | Watch 4

图16-2 仿真例程16-2时的观察窗口

按照图 16-3 所示设置断点并运行程序到第 1 个断点处。然后小心地单步执行程序，注意观察每行的动作。图中用注释“Do apples”和“Do pears”分别标出 2 个程序段。每个程序段的动作实际上就是为数组 **list** 装载 1 个数字和 1 个商品类型，即 apple 或 pear。

```
//Do apples
    counter = 0;                         //set index to list
B   list[counter] = number;              //indicate number of items
    while (item1[counter] != 0)          //indicate type of item
    {list[counter+1] = *(pntr1+counter);
    counter = counter + 1;
    }
    if (number >1)list[counter+1] = plural; //set the item to plural
    else list[counter+1] = 0;            //return item to single

//Do pears
B   counter = 0;                         //set index to list
    list[counter] = number;              //indicate number of items
    while (item2[counter] != 0)          //indicate type of item
    {list[counter+1] = *(item2+counter);
    counter = counter + 1;
```

图 16-3 例程 16-2 的建议断点位置

在初始化 **counter** 之后，使用下述代码向数组中第 1 个单元传送数字：

```
list[counter] = number;                //indicate number of items
```

上述代码使用了最简单的数组访问形式，方括号中的数字指示数组元素。上例中，将 **number** 赋予数组中的第 1 个元素。

435

接下来程序使用了 **while** 循环。由于字符串以空字符作为终止符，因此可以相应地进行检测。下列代码使用了“不等于”运算符：

```
while (item1[counter] != 0)              //indicate type of item
```

只要字符非空，就用下述代码将字符串 **item1** 中的字符传送给数组 **list**：

```
{list[counter+1] = *(pntr1+counter);
```

列表元素依然由方括号中的索引所决定。此处在 **counter** 值上加上了偏移量 1，这是因为它的首个元素已经赋过值了。为这些元素所赋的值是 **pntr1** 加上偏移量 **counter** 所对应的间接值。

下面对苹果数是否大于 1 进行测试。如果大于 1，需要在商品类型上加上 s，否则插入空格。

```
if (number >1)list[counter+1] = plural;  //set the item to plural
else list[counter+1] = 0x20;  //return item to single, with ASCII space
```

注意在程序中的 Pears 段中，传送字符时采用了不同的方式。此时，将数组名作基址，加上偏移量 **counter**，以此来确定 **item2** 字符串中的数组元素。最终将该计算地址所对应的间接值传送给 **list**。

```
{list[counter+1] = *(item2+counter);
```

在循环结尾处将 **number** 的值增加 1。在赋值语句的右边，通过指针所对应的间接值(即 **number**)来访问 **number**。

```
number = *pntr2 + 1;
```

单步执行程序，直至你理解每行代码的功能为止。然后在断点间运行程序，观察 **list** 数组的内容在每次循环中是如何更新的。

上面有一个现象可能会令人感到奇怪：从观察窗口可以看出，程序中定义的字符串都放置在数据存储器中。一般情况下，我们希望它们出现在程序存储器中。在第 17 章中，我们将回到这个问题，在那里将探讨如何控制字符串和其他常数所在的存储器类型。

16.4 使用 I²C 外围设备

第 8 章已经提到，I²C 是一种有效的串行通信标准，但使用起来略显复杂。C18 编译器库中提供了一些非常有用的函数，利用它们可以方便、可靠地实现大多数 I²C 功能。参考文献 14.3 列出了不下 15 种 I²C 所需函数，其中一些函数的功能类似。表 16-3按照可能的使用顺序给出了一些例子。

表 16-3 I²C 库函数示例

函 数	动 作
OpenI2C()	将 SSP 模块配置为 I²C
StartI2C()	产生 I²C 启动条件
WriteI2C()	向 I²C 写单个字节
ReadI2C()	从 I²C 读单个字节
StopI2C()	产生 I²C 停止条件

16.4.1 I²C 例程

例程 16-3 给出了一个应用微控制器 I²C 性能的简单程序，其中使用了表 16-3 中的函数。程序是为 Derbot AGV 所编写的，首先向手动控制器发送一个字符，然后发送一个字符串。

436

例程 16-3 使用 I^2C 向 Derbot 手动控制器发送字符和字符串


```
/**********************************************************************
Dbt_I2C_test_c
Sends single character, and then string, periodically on I2C.
Set up as Master.
Files c018i.o and p18f242.lib are included by the Linker Script.
TJW 12.11.05                                              Tested 4.12.05
***********************************************************************
Clock is 4MHz.
Configuration Word all default, except: crystal oscillator (HS),
power-up timer on, brown-out detect off, WDT off, LV Program disabled*/

    #include <p18F242.h>
    #include <i2c.h>             //header file for I2C
    #include <delays.h>          //header file for delays .
    #define slave_addr1 0xA4   //Adress of Derbot Hand Controller I2C node.

//Function Prototypes. Library function prototypes are found in Header file.
    void diagnostic (void);

//constants & variables
    unsigned char message[] = " Derbot";
    unsigned char *i = &message[0];  //pointer to message[]
    char loop_cntr = 0;
/*********************************************************************
This is main function.
*********************************************************************/
    void main (void)
{
//Initialise Ports
    TRISA = 0b00001011;         //ADC channels set as inputs
    TRISB = 0b11001000;
    TRISC = 0b10011000;         //I2C bits are both set as ip
//Switch all outputs off
    PORTA = PORTB = PORTC = 0;
//call diagnostic function (flash leds)
    diagnostic();
//Initialise I2C
    OpenI2C (MASTER, SLEW_OFF);
    SSPADD = 0x07;              //set up 125kHz baud rate
loop:
//Send single character
    i = &message[0];
    StartI2C();                 //send start condition
    WriteI2C (slave_addr1);     //send address word, function waits until
                                  //write is complete
    loop_cntr = loop_cntr| 0x30;  //convert counter to ASCII
    WriteI2C (loop_cntr);
    loop_cntr = loop_cntr&0x0F;  //retrieve counter from ASCII
    StopI2C();                  //send stop condition
    Delay10KTCYx (100);
//Send a string
    StartI2C();                 //send start condition
```

```
    WriteI2C (slave_addr1);   //send address word, function waits until
                                   //write is complete
    while (*i)                //Test for null character
    {
    WriteI2C (*i);
    i++;
    Delay1KTCYx (5);          //delay needed for hand controller
    }
    StopI2C();                //send stop condition
    Delay10KTCYx (100);
    loop_cntr++;
    if     (loop_cntr == 10) loop_cntr = 0;
    goto loop;
}                             //end of main
//Diagnostic: switches leds on for 1s (Tcy = 1us)
void diagnostic (void)
...
(same as diagnostic function, Program Example 15.1)
...
```

438

程序起始部分的几行代码指明了程序所包含的头文件，并将手动控制器从动节点地址定义为 $A4_H$（详见10.8节）。程序还声明了将要发送的信息中的字符串，然后为其指定一个指针。

I^2C 端口的部分初始化由函数 **OpenI2C()** 实现，该函数有2个参数（详见参考文献14.3），用于确定运行模式并选择回转速率。注意，此时仍需要通过写 **SSPADD** 寄存器来设置波特率。然后通过 **StartI2C()** 函数在串行连接上施加启动条件，作为 I^2C 信息传输的开始。紧接着通过向 **WriteI2C()** 函数传递参数来发送地址字节。它将被传输字中的 $R/\overline{W}$ 位设置为0，如图10-13所示。然后将循环计数器值转换为ASCII码，并再次使用 **WriteI2C()** 函数进行发送。最后使用 **StopI2C()** 函数终止信息传输。

字符串的发送方式与上述方法基本类似，只是加入了少许改进。仍然使用 **StartI2C()** 函数来启动 I^2C 信息传输，并发送从动器地址。然后建立 **while** 循环，条件如下：

```
while (*i)
```

其中 **i** 是指向字符串的指针，***i** 表示字符串中的某个元素。注意，字符串中的最末元素为空字符。循环重复进行，直至到达字符串结尾，此时将跳出循环。

16.4.2 使用++和--运算符

在接近程序结尾的地方，会发现索引 **i** 和 **loop_cntr** 都使用了++运算符。如表A6-5所示，该运算符用于对所涉变量进行增1操作。因此：

i++；看起来就等价于 i=i+1；

但是，它们之间存在重要的差别。当将该操作符放在变量前面时，表示预递增。放在变量后面表示后递增。在该例程中，上述差别并不明显。但是，可以通过下面的两个例子加以理解：

```
index = 4;                    index = 4;
new_val = index++;            new_val = ++index;
```

439

在上述左侧例子中，**new_val** 取值为 4，然后增加 1。而在右侧例子中，**index** 先进行（预）增 1 操作，那么 **new_val** 就取值为 5 了。

递减运算符——的使用方法与上述方法完全相同。

16.5　格式化显示数据

我们到此已经了解了如何通过 I²C 连接向 LCD 显示器发送字符和字符串，但仍然需要生成一些有意义的数据用于显示。本节将开发 Derbot"寻光"程序（见例程 16-1），将光敏电阻读数显示在手动控制器显示屏上。

16.5.1　例程概览

在例程 16-1 中，从 ADC 读到的转换值为 10 位数字。要将该数值转换为字符串，需要首先转换为 BCD 码，然后再转换为 ASCII 码。例程 11-2 是使用汇编语言来实现的，但确实非常繁琐。那么使用 C 语言是否会好点呢？

上述问题的答案是完全肯定的，C 语言中有很多函数可以将数据从一种格式转换为另一种格式。表 14-3 给出了一些示例。在这里，我们需要一个函数能够处理 10 位 ADC 输出，并能将其转换为字符串。使用函数 **itoa**（读作 i-to-a，整型到 ASCII）可以方便地实现上述功能。

例程 16-4 给出了程序 **light_seek_&_disp** 的一部分，完整程序在本书附属资源中。给出的程序片断是对例程 16-1 的扩展。其中编写了 2 个新函数。一个是 **disp_int()**，它对 ADC 输出值进行格式化，然后在 I²C 连接上发送结果字符串。另一个是 **send_space()**，它只是向显示屏发送一些空格，用以优化 LCD 上的信息布局。

主循环每迭代执行 10 次，才显示一次数据。如果显示频率过快，将使数据产生不必要的闪烁。在主循环中插入了循环计数器 **loop_cntr**，循环每执行一次就增加 1。程序 3 次调用函数 **disp_int()** 来显示数据，分别对应 3 个 LDR。前两个显示结果放置在两行显示屏的第 1 行中，而在后侧 LDR 显示位置前后都插入空格，使其居中放置在第 2 行。

例程 16-4　格式化数据用于 LCD 显示

```
...
    #include <stdlib.h>           //for itoa function
    #include <string.h>           //for strlen function
...
(this program section is placed within the main program loop, after the ADC
values have been read)
//Display ldr values every 10 loops
    if (loop_cntr == 10)
    {
```

440

```
      disp_int (ldr_left);        //display left ldr value on lcd
      disp_int (ldr_rt);          //display right ldr value on lcd
      send_space (2);             //centre rear display
      disp_int (ldr_rear);        //display rear ldr value on lcd
      send_space (2);             //fills second line, forcing line feed
      loop_cntr = 0;
      }
...
...
/*********************************************
Display Functions
*********************************************/
//Converts an integer to string,and sends to lcd via I2C,filling with spaces to
ensure 4 digits are always sent
      void disp_int (int op_int)
{
      char disp_val[6];           //will hold the string representation of
                                   //any ldr value
      char *disp_val_ptr;         //pointer to disp_val[]
      char space_no;              //number of spaces to be inserted
      disp_val_ptr = &disp_val[0];
      itoa (op_int, disp_val_ptr);//first convert to a BCD string
      space_no = 4 - strlen(disp_val_ptr); //find how many spaces needed
//Now send the message
      StartI2C();                 //send start condition
      WriteI2C (slave_addr1);     //send address word
      while (space_no)            //fill up with leading spaces
      {WriteI2C (' ');
      Delay1KTCYx(5);             //little delay needed by hand controller
      space_no--;
      }
//send the string
      while (*disp_val_ptr)
      {
      WriteI2C (*disp_val_ptr);
      disp_val_ptr++;
      Delay1KTCYx(5);             //delay needed for hand controller
      }
      StopI2C();                  //send stop condition
}

//Sends space to lcd via I2C
      void send_space (char space_no)
{
      StartI2C();                 //send start condition
      WriteI2C (slave_addr1);     //send address word,
//send space
      while (space_no)
      {
      WriteI2C (' ');
      Delay1KTCYx(5);             //delay needed for hand controller
      space_no--;
      }
      StopI2C();                  //send stop condition
}
```

16.5.2 使用库函数实现数据的格式化

让我们仔细分析一下例程 16-4 中的函数 **disp_int()**，它包含一些重要的特性。首先注意到，在程序开头对数组、指针和字符变量进行了声明。这些变量仅在函数的执行期内存在。数组中包含 5 个单元，由于 10 位二进制数的十进制表示(1024_D)最多包含 4 位，因此数组中的第 5 个字节用于存放终止空字符。向函数传递的参数为 **op_int**。然后定义指针并指向该数组的起始字节。

接着调用函数 **itoa()**，它的函数原型如下：

```
char * itoa (int value, char * string);
```

其中，**value** 是将要被转换的整数，**string** 是用于存放结果的字符串，函数返回值是指向字符串的指针。它的具体实现如下：

```
itoa (op_int, disp_val_ptr);//first convert to a BCD string
```

可以看出，**op_int** 是被转换的变量，通过使用预先声明的指针 **disp_val_ptr** 将转换得到的字符串放置在数组 **disp_val** 中。

此时似乎可以直接将字符串送到显示屏上，但是请注意，该字符串的长度可能是 1～4位内的任意数。为了确保它在显示屏上占据相同的位置，有必要先找到它的长度。可以使用通用软件库中的 **strlen()** 函数来实现。该函数测量字符串长度并将该值返回。函数原型如下：

```
size_t strlen(const char *string);
```

这里，**string** 是被测字符串，返回值 **size_t** 包含了字符串的长度。程序利用该函数计算 **space_no** 的取值，也即需要发送的空格数，从而使显示的数字达到 4 位。这些空格在数据之前发送，接着发送字符串，最后程序终止。

16.5.3 程序分析

上述例程将先前的测光程序(例程 11-2)和寻光程序融合在一起。可以按照 16.2.4 节所述，采用类似方法进行程序仿真。通过在 **ldr_left**、**ldr_rt** 和 **ldr_rear** 中插入实验值，观察字符串的转换过程，以及对字符串长度的测试过程。如果让程序在Derbot 上运行，也会得到满意的结果，Derbot 的动作与显示屏上的数据可以非常清晰地解释程序的执行过程。

442

小结

本章主要介绍了如何在嵌入式系统下使用C语言来实现数据的采集和整型数的使用。要点如下所示。

- 18FXX2 ADC和I^2C串行端口可以直接使用库函数来驱动。
- 数组、字符串以及相关指针提供了强大的数据集处理方式。在理解C语言对这些对象的处理方式时需要多加留意。
- 利用一些有用的库函数可以对数据字符串进行操作。它们可以非常方便地格式化数据并直接进行显示。

443

第17章 深入学习C语言编程和更丰富的C语言编程环境

现在,我们已经对在PIC®控制器上编写一些简单的C语言程序有一定的信心了。但是,在某些知识上我们还是存在缺陷,比如中断的使用。对它的学习会使我们更深入地理解如何使用C语言进行编程,实际上这也超出C语言本身的范畴。由于程序变得越来越复杂,了解更丰富的编程环境是非常有用的,例如程序中连接或者包含的一些文件。到目前为止,我们只是使用这些文件却很少知道它们是如何起作用的。

因此,这一章有2个目的。第1个就是学习C语言方面的知识,以便我们可以更接近硬件底层来编写程序,这些知识包括使用内嵌汇编和中断。第2个是学习更丰富的C语言编程环境。为了达到这个目的,我们需要了解C语言本身的一些特定内容。

在本章中你将学到:

- ☐ 内嵌汇编的使用;
- ☐ 中断的使用;
- ☐ 进一步了解数据定义和数据的存储类型以及如何控制存储器的使用;
- ☐ 经常连接到应用程序中的一些文件,包括头文件和启动文件;
- ☐ 连接器和连接器脚本。

如同前面其他几章一样,本章也使用了大量实例。

17.1 深入学习C语言编程和更丰富的C语言编程环境

尽管本章旨在介绍C语言的一些更加深入的专门知识,但是不会涉及C语言编程本身的一些技巧。相反,我们会看到使用C语言进行底层硬件编程的限制,这个限制在本章将通过中断或者特殊存储器映射的方法解决。我们也会学习一些源代码之外的连接文件或包含文件以及把它们连接起来的工具——连接器。

为了联系上面这些不同的内容,我们将从 **c018i.c**(这个文件启动本书中所有的C语言程序)这个启动文件中截取出许多例子来学习。它包括许多有趣的编程特性。通过学习这些特性,我们同时将学习这个重要的程序。这个启动程序不太容易理解,但并不需要完全看懂它。建议把这个程序打印出来,在学习本章时随时查阅。这个程序的代码比较少——打印大约需要两页半纸。如果在MPLAB®中仿真本书中的任何一

个C语言程序，当仿真开始时，这个程序的源代码会自动弹出。另外，在C18安装文件夹的**mcc18\src\traditional\startup**目录下也可以找到它。

17.2 插入汇编

尽管在嵌入式环境中C语言非常高效，但是仍然会存在一些最好使用汇编语言进行编程的情形。这些情形包括：

- 一些特定的指令引发与处理器相关的动作，而C语言中没有等价的语句时——PIC系列微处理器中包括**SLEEP**或**CLRWDT**指令；
- 对程序执行的时间要求非常特殊，程序员需要直接控制一个特定程序段的编写时；
- 一个程序段需要高效执行，程序员希望尽可能以最高效的方式来编写它时。

因此，在C语言程序中能够编写汇编语言是非常有用的。在C语言中编写汇编程序的方式称为*内嵌汇编*(in-line assembler)。它提供了另外一种编写汇编程序的方式，这种方式区别于整个源文件都使用汇编语言然后使用构建过程把它连接到主程序的方式，如图14-1所示。

MPLAB C18编译器允许在C语言程序中嵌入汇编程序。可以嵌入一条汇编语句乃至一个完整的汇编程序块。你可能会认为C编译器将调用MPASM™(常规的MPLAB汇编器)来编译内嵌汇编代码。但是，事实并非如此。C18编译器本身有一个内部汇编器来编译这些内嵌汇编代码。

C18内部的汇编器在很多方面区别于常规MPLAB汇编器。主要区别如下：

- 内嵌汇编语句必须处于标示符**_asm**和**_endasm**之间；
- 不能使用汇编伪指令；
- 必须使用C语言或C++的注释风格；
- 操作数不能省略，即操作数必须全部指定；
- 必须使用完整格式的表读写指令(参看表A5-1的最后部分)；
- 默认的数制是十进制；
- 符号定义使用C中的基数计数法；
- 标号必须以冒号结尾。

例程17-1给出了一个内嵌汇编代码的例子。尽管一眼看上去它似乎像常规的汇编代码段，但是实际上这个代码段包含了至少6种不同的内嵌汇编特性。所有这些特性都在图中进行了标注，可直接归类到上述列表中。这个例子是从启动文件**c018i.c**中截取的。

要谨慎地使用内嵌汇编，特别是当汇编代码块长度是任意时。正如我们所知，编写汇编要遵循的规范要比C语言少。因此，程序员在编写汇编程序段时，很容易违反C语言编程的规范(甚至自己都没有觉察到)，从而就破坏了宿主程序——C程序的结构。

作为初学者,不建议编写会修改到C程序中声明的变量或者函数的内嵌汇编。如果需要编写一个较大的汇编语句块,最好在一个单独的文件中来编写,然后使用MPASM汇编器来汇编这个汇编文件,最后把它连接到主程序。这种方式有助于维持存储器映射、变量使用以及函数调用的顺序。

例程 17-1　启动文件 c018i.c 中的代码片段

```
  _asm                                          ← 内嵌汇编的开始
     // determine if we have any more bytes to copy   ← C语言风格的注释
     movlb curr_byte
     movf curr_byte,1,1
copy_loop:                                      ← 标号以冒号结尾
     bnz 2   // copy_one_byte
     movf curr_byte+1, 1, 1
     bz 7    // done_copying

copy_one_byte:                                  ← 完整格式的TBLRD*+
     tblrdpostinc
     movf TABLAT, 0, 0
     movwf POSTINC0, 0

     // decrement byte counter
     decf curr_byte, 1, 1                       ← 指定默认操作数的值
...
  _endasm                                       ← 内嵌汇编的结束
```

17.3　控制存储器分配

使用高级语言编程的一个好处之一就是程序员不需要考虑存储器映射以及如何分配存储空间。C编译器会(如果我们想让它做的话)替我们完成几乎所有这些工作。当然,在编译的某个阶段,需要给编译器输入与编译目标计算机的存储器相关的一些信息。这个过程隐含在连接器脚本中,17.10节会讲到。

但是在嵌入式环境中我们需要在使用高级语言编程的某些情形下重新控制存储器的分配。这些情形包括一些与硬件相关的行为,例如中断或者配置位的处理,以及对存储器映射进行优化,这是一个较大的话题。C18编译器中有很多控制存储器分配的方法。有一些方法非常复杂,只有经验丰富的程序员才会使用它们。但是所有的程序员必须要了解一些简单的方法。下面我们就来学习它们。

446

17.3.1　存储器分配伪指令 pragma

我们已经在14.2节提到了预处理机的伪指令概念。一个特殊的伪指令是 **#pragma**。它允许部分C语言代码定制成特殊编译的——每次使用 **#pragma** 时,紧随其后的程序代码将使用特殊方式来进行编译。

C18 编译器有 4 个用于控制存储器分配的 pragma 伪指令。它们修改段(即一个特殊的被指定的存储器块,编译器在其中存放数据)的内容。这 4 个 pragma 如表 17-1 所示。每个都有许多不同的参数选项,参考文献 14.2 中给出了它们的完整格式。这些完整格式的 pragma 非常复杂,因此在这里我们只学习它们的部分细节。

常用的 **#pragma** 是表中的第一行,它通常的格式如下:

#pragma code(段名称)(=地址)

它的作用类似于汇编程序中的 **org** 伪指令,必要时使用它来指定程序代码在存储器中的位置。两个括号中的参数是可选的。表中有一个 pragma 伪指令用来启动我们编写的每一个 C18 程序,使程序执行到 **c018i.c** 的起始处。开始几行如下所示:

```
#pragma code _entry_scn=0x00
void _entry (void)
{
_asm goto _startup _endasm
}
#pragma code _startup_scn
void
_startup (void)
{
...
```

这里特定格式的 **#pragma code** 伪指令携带一个段名 **_entry_scn** 和一个地址 0x00 参数。下面几行又使用了携带有一个段名 **_startup_scn** 的 pragma 伪指令,但是这一次没有指定地址参数。因此,这个伪指令允许连接器自身来设置地址。**_entry_scn** 和 **_startup_scn** 被 C18 编译器指定为保留字,前一个用来定位复位向量,后一个用来保存启动代码。

表 17-1 用于存储器分配的 Pragma 伪指令

Pragma	作 用
#pragma code…	定位程序代码在程序存储器中的位置
#pragma romdata…	定位数据在程序存储器中的位置
#pragma udata…	定位未初始化的用户变量在数据存储器中的位置
#pragma idata…	定位初始化的用户变量在数据存储器中的位置

17.3.2 设置配置字

由于 PIC 18 系列微控制器有大量的配置位,在程序中需要使用很多条语句来设置它。C18 的 3.0 版编译器有一个简单的方法来设置配置字:使用 **#pragma config** 伪指令。实际中配置字的设置是与处理器相关的。即使对于一个特定的微控制器,用于设置配置字的语句也是非常多的。因此,这里我们不列出所有的配置,但是可以在参考文献 17.1 中找到 PIC 18 系列所有微控制器的配置字设置。

例程 17-2 列出了适合 AGV-18 的配置字设置。回顾表 12-4 来核对一下这些配置

字的设置是很有用的。每一行 #**pragma config** 用来设置一个配置字，这些伪指令的格式可以查看参考文献 17.1。未定义的配置位将使用默认值。

例程 17-2 AGV 中的配置字设置

```
#pragma config OSC = HS, OSCS = OFF  //oscillator type is HS, oscillator switch is off
#pragma config PWRT = ON, BOR = OFF  //power-up timer is on, brown-out detect is off
#pragma config WDT = OFF             //watchdog timer is off
#pragma config STVR = ON, LVP = OFF  //Stack overflow reset enable is on,
                                     //low voltage programming is off
```

一旦在程序中使用 pragma 伪指令来设置配置字，它将覆盖掉 MPLAB IDE 软件界面上 **Configuration Bits** 窗口中的任何配置字的设置。在构建项目期间，编译器将根据程序中的配置字来设置微控制器的配置字。你可以在配置位窗口中设置“错误”的配置字，然后构建项目，在实际中检验一下配置字的设置方式。然而，如果配置字中的某一位没有在源程序中设置，但是在 **Configuration Bits** 窗口中通过特殊方式进行了设置，那么构建过程不会将该配置位设置为默认值。正如 7.11.3 节所述，在 MPLAB 软件界面中单击下拉菜单 **Configure > Settings> Programmer Loading**，然后选择 **Clear configuration bits upon loading the program**。这种方式可以保证当程序下载时，窗口中配置字的设置会被清除，只有程序中定义的配置位才会被下载。

17.4 中断

在 C 语言编程环境下，关于中断有很多需要解决的问题。当使用中断时，我们正在和底层的硬件打交道，但是高级语言趋使我们远离底层硬件。为了保证 PIC 18 系列微控制器的中断能够正常工作，必须按顺序执行许多特殊而重要的操作步骤。中断必须被启用并设置成需要的优先权。中断服务程序必须放置在程序存储器中正确的地址（注意 PIC 18 系列微控制器有 2 个中断向量表），并且在中断服务程序中要保护上下文。你可以查看图 12-7 中所示的内容以及相应的对正确处理中断的描述。

17.4.1 中断服务程序

当中断发生时，微控制器将调用中断服务程序，并以一条中断返回指令来结束中断服务程序。除此之外，C 语言中的中断服务程序类似于普通的 C 语言函数。在中断服务程序中可以声明局部变量和访问全局变量。但是中断服务程序访问的全局变量一定要声明为 **volatile** 类型。这种类型的变量表明它的值可以在正常的程序操作流程之外被修改。由于可以在任何地方调用中断服务程序，所以它不允许携带任何入口参数和返回值。

448

17.4.2 定位和识别中断服务程序

C18 编译器使用了一些 pragma 伪指令。首先用它来定位中断服务程序在复位向

量中的起始地址，之后用它来区分中断服务程序和普通的函数。

正如复位向量一样，C18 编译器不会自动启动存放在程序存储器中高优先级或者低优先级的中断服务程序。这就需要使用已经提到的 **#pragma code** 伪指令。它用来在复位向量表中定位中断服务程序的起始地址。

在程序中使用 pragma 伪指令定义中断服务程序。有 2 种中断服务程序的定义格式，如下所示。

- **#pragma interrupt** function_name(**save**=save_list)。这个伪指令声明了一个高优先级的中断服务程序 function_name。快速寄存器栈（参见 12.6.3 节）用来保存最小上下文——**STATUS**、**WREG** 和 **BSR** 寄存器。中断服务程序以快速返回方式结束。
- **#pragma interruptlow** function_name(**save**=save_list)。这个伪指令声明了一个低优先级的中断服务程序 function_name。软件栈用来保存最小上下文。这种方式减慢了微控制器对中断的反应速度。中断服务程序以正常返回方式结束。

如果需要进行的上下文保护比最小上下文多，那么可以在定义任何优先级的中断服务程序的伪指令中指定需要保存的寄存器——在 pragma 的 **save** 参数段进行指定。

17.5 使用溢出中断的例子——闪烁 AGV 上的 LED

例程 17-3 使用了 18F242 的溢出中断来闪烁 AGV 上的 LED。这个看似简单的程序将启发我们继续进行一些更深入而且实用的 C 编程。

例程 17-3 使用 Timer 0 中断的"闪烁 LED"程序

```
/*************************************************************************
Flashing LEDS
Flashes Derbot LEDs, driven by Timer 0 interrupt on overflow.
Demonstrates: Use of Timer 0 peripheral, Interrupts, and inline assembly.
TJW 30.10.05
*************************************************************************/

#include <p18F242.h>
#include <timers.h>
#pragma config OSC = HS, OSCS = OFF  // HS oscillator, oscillator switch off
#pragma config PWRT = ON, BOR = OFF  //power-up timer on, brown-out detect off
#pragma config WDT = OFF             //watchdog timer off
#pragma config STVR = ON, LVP = OFF  //Stack overflow reset enable on,
                                         //low voltage programming off
//function prototype, repeated for info
void timer0_isr (void);

unsigned char counter = 0;
```

```
//Define the high interrupt vector to be at 0008h
#pragma code high_vector=0x08
void interrupt (void)
{
  _asm GOTO timer0_isr _endasm  //jump to ISR
}
#pragma code  //Return to default code section

//Function timer0_isr specified as high-priority ISR
#pragma interrupt timer0_isr

//timer0_isr function. No transfer of parameters, as required by ISRs
void timer0_isr (void)
{
  counter = counter + 1;
  PORTC = counter<<5;        //Shift Counter left, and move to PORT C
  INTCONbits.TMR0IF = 0;     //Clear TMR0 interrupt flag
}
void main (void)
{
  //Initialise
      TRISC = 0b10000000;
      PORTC = 0;            //Switch outputs off

/*Initialise TMR0: interrupt enabled,16-bit operation, internal clock,
prescaler divide by 4, hence (with 4MHz clock)period of 1usx64kx4 = 262ms*/

  OpenTimer0 (TIMER_INT_ON & T0_SOURCE_INT & T0_16BIT & T0_PS_1_4);
  INTCONbits.GIE = 1;       //Enable global interrupt

  while (1)   //Await interrupts
    {
    }
}
```

你可以看到主程序是很短的——一旦初始化完成，所有程序活动都在中断服务程序中进行。主程序执行的全部动作包括初始化端口C为全0、初始化Timer 0(见下一节)、设置全局中断启用、最后进入一个不会结束的循环来等待中断发生。

17.5.1 使用 Timer 0

我们已在15.5.1节遇到了适用于PIC 18系列定时器的库函数。Timer 0硬件本身在13.3.1节已经讲述，并且可从图13-2中很明显地看到它有多种工作模式。如果在任何程序中使用了定时器，必须在程序中包含定时器的头文件 **timers.h**，正如这里所看到的。

450

在程序中发现 **main** 函数的初始化部分使用了 **OpenTimer 0** 函数，如下所示：

```
OpenTimer0 (TIMER_INT_ON & T0_SOURCE_INT & T0_16BIT & T0_PS_1_4);
```

启动定时器的配置数据由函数的参数指定。参考文献 14.3 对这些参数的取值范围进行了描述。这个函数用于启用定时器的溢出中断、设置定时器的时钟源为内部振荡器、配置定时器为 16 位操作模式(对比于 8 位模式)、设置预分频比为÷4。一旦设置完毕,定时器开始自由运行并且产生一系列需要微控制器响应的中断。在这种设置下,计数器需要 65536 个周期才能从 0 递增到最大值 65535 并且输入时钟的周期为 4μs。因此,中断的间隔时间为 65535×4μs,即 262.144ms。

17.5.2 中断的使用和中断服务程序的动作

这个例程仅仅使用了 Timer 0 中断并且没有设置中断的优先权,所以默认使用高优先权的中断向量。正如我们刚提到的,在程序中需要使用伪指令 **#pragma code** 来设置中断向量。为了设置高优先权的中断向量(见图 12-6),使用了下面的语句:

```
#pragma code high_vector = 0x08
```

这条语句指定了紧随其后的代码存放在程序存储器中以 08_H 开始的地址单元。这个地址由程序员命名为 **high_vector**。它并不是 C 语言中的保留字,因此也可以选择其他名称。另外,下面的语句用来设置低优先级的中断向量:

```
#pragma code low_vector=0x18
```

这条语句指定紧随其后的代码存放在程序存储器中以 18_H 开始的地址单元。这个地址由程序员命名为 **low_vector**。

当正确设置中断向量之后,下面紧跟着的一行内嵌汇编代码语句是用来强制将程序跳转到中断服务程序的:

```
_asm GOTO timer0_isr _endasm //jump to ISR
```

为了允许编译器再次定位代码的位置,使用下面的伪定义"取消"前面 pragma 的定义:

```
#pragma code
```

它将代码的定位恢复到默认位置。

中断服务程序 **timer0_ISR** 的函数原型出现在程序清单的开始处。中断服务程序的动作很简单。首先,**counter** 的值加 1。然后将它左移 5 次,之后赋值给端口 C。这些操作的效果是使用 **counter** 的低 2 位来控制连到端口 C 的第 5 位和第 6 位的 LED。然后 Timer 0 的中断标志被清零。

这个关于中断的简单例程仅仅使用了一个中断,并且没有设置 PIC 18 系列微控制器的中断优先权。例程 19-6 中会讲到一个使用 2 个中断并设置中断优先权的例子。

17.5.3 仿真闪烁 LED 程序

不管你是否将程序下载到实际的硬件中,仿真例程 17-3 都是很有意思的。使用随

书附属资源中程序的源代码来创建一个项目。构建项目之后，使用 MPLAB 仿真器来仿真这个程序。打开 Watch 窗口，并且在窗口中显示寄存器 **PCL**、**counter**、**PORTC** 和 **INTCON**，如图 17-1a 所示。快速单步执行完程序的开始部分，并且注意 **OpenTimer0** 函数是如何出现的。一旦程序执行到 **main** 函数，并且在用户初始化过程完成之后，程序进入一个不会结束的 **while** 循环来等待中断发生。

Watch

Add SFR ADCON0 Add Symbol ADCON0

Address	Symbol Name	Hex	Decimal	Binary
008A	counter	02	2	00000010
0FF9	PCL	08	8	00001000
0FF2	INTCON	24	36	00100100
0F82	PORTC	40	64	01000000
0FD6	TMR0	0001	1	00000000 00000001

Watch 1 Watch 2 Watch 3 Watch 4

(a) Watch窗口

```
#pragma code high_vector=0x08
void interrupt (void)
{
B   _asm GOTO timer0_isr _endasm  //jump to ISR
}
```

(b) 断点位置

图 17-1 仿真闪烁 LED 程序时建议的设置

现在，你通过在 Watch 窗口中设置 Timer 0 的中断标志位（**INTCON** 寄存器的位 2）为 1 来强制触发一个中断。几次单步执行之后，高优先权的中断向量中的内嵌汇编语句将程序跳转到中断服务程序处。通过连续地单步执行程序，你可以观察到中断服务程序的动作。

现在我们来检验一下定时器中断的动作。首先，在中断服务程序的入口地址处设置一个断点，如图 17-1b 所示。如果程序在任何时候执行到这里（即每一次发生定时器中断的时候），它都将停下来。在工具条中，选择 **Debugger>Settings>Osc/Trace** 并设置振荡器的频率为 4MHz。从 Debugger 下拉菜单中打开 Stopwatch 窗口，如图 17-2 所示。

452

不管程序执行到何处，现在将程序运行到断点处。然后清零 Stopwatch 窗口中的数值，再次运行。过一段时间，程序应该再次停到断点处。Stopwatch 窗口中显示的时间应该是 262.144ms，与图 17-2 中所示完全相同。这也确认了理论计算得到的每次溢出之间的时间间隔。同时，Watch 窗口的值应该类似于图 17-1a 中所示。从图中可以看出，Timer 0 刚溢出，准备再次从零开始递增。程序计数器的低字节（**PCL**）被设置为高优先级的复位向量 08_H，并且定时器的中断标志（**INTCON** 寄存器的位 2）被设置为 1。

如果构建了 AGV-18 硬件，就可以在硬件上运行这个程序来闪烁 LED。

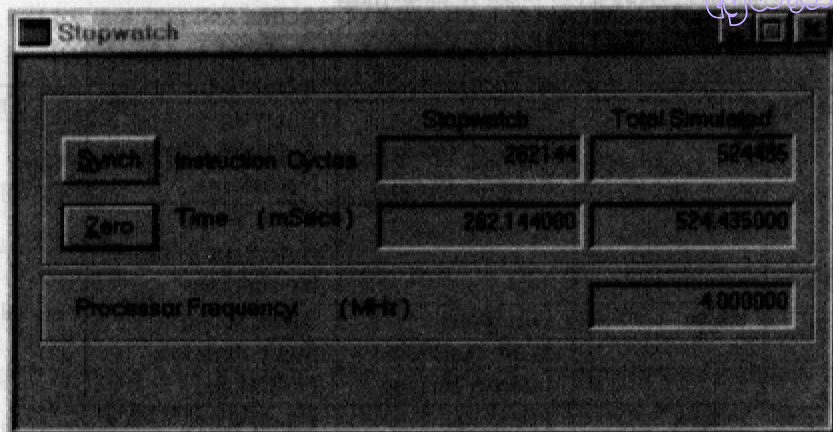


图 17-2 仿真“闪烁 LED”程序时的 Stopwatch 窗口

17.6 变量的存储类型及其应用

本章后续部分主要学习更丰富的 C 语言编程环境——主要由一些文件组成。其中重要的文件有 3 个，每个程序都会使用到它们：微控制器的头文件、启动文件（start_up 文件）和连接器脚本。为了理解它们，有必要研究一下 C 语言其他方面的内容。这里我们首先学习 C 语言中变量的存储类型。

17.6.1 存储类型

正如我们前面看到的，C 语言谨慎地控制对数据的使用——数据如何声明以及怎么被使用。这样做是有必要的，部分原因是由于 C 程序可以编写的很复杂——它由不同的文件和函数构成、由不同的程序员在不同时间进行修改、在编辑的不同阶段被保存。比如，函数和数据可能在一个文件中被声明，但是需要在其他文件中访问它们。

因此，C 语言中数据的一个特性是它的*存储类型*（storage class）。它定义了在代码块、函数或者文件中所声明的变量的状态。关于变量的存储类型，表 A6-3 给出了 4 个关键字。 453

C18 编译器只使用了其中的 3 个修饰关键词：**auto**、**static** 和 **extern**。这些关键词有点奇特，似乎并不能起多大作用。即使我们能够意识到 **auto** 有自动暂时的含义、**extern** 能将变量声明为外部程序可用，但似乎仍然很难理解它们真正的含义。*自动*一词的用法是借用其他计算机语言，它意味着变量在一个特定的函数内以一个特定的目的产生，但是不会在其他时间存在。但是，*静态*意味着可以一直存在。*外部*意味着变量可以持续存在并且能被任何函数访问。

回到存储类型，它确定了一个变量的 3 个方面：*可见性*、*生存期*和*连接*。下面依次来对它们进行研究。表 17-2 总结了这 3 个方面的所有组合。当然，其中有一些的使用

非常广泛，在本书的例程中我们只选择几种来讲解。

表 17-2 变量存储类型：修饰关键词和声明位置的作用

存储类型修饰关键词	在所有函数外声明	在函数内声明
none	文件可见性 静态生存期 外部连接	块可见性 自动生存期 无连接
auto	—	块可见性 自动生存期 无连接
static	文件可见性 静态生存期 内部连接	块可见性 静态生存期 无连接
extern	文件可见性 静态生存期 外部连接	块可见性 静态生存期 外部连接

17.6.2 可见性

一个变量的可见性决定了可以在程序的哪个部分访问它。以下是 2 个可见范围。

- 块可见。变量只能在声明的块内访问，而且是在声明语句之后。这种类型的变量被称为局部变量。在程序的不同块内局部变量的命名可以相同，这些名称相同的变量之间没有任何关系。
- 文件可见。变量只能在被声明的文件中访问，而且是在声明语句之后。变量必须在所有块之外被声明。

17.6.3 生存期

一个变量的生存期可以是下面 2 种类型之一。

454

- 自动存储生存期。一个 **auto** 类型变量是声明在一个代码块内的，每次程序进入这个块时，都将重新产生它。当块运行结束时，变量也将消亡，且它占据的内存被释放。使用关键字 **auto** 来定义具有这种类型生存期的变量。由于在块内声明变量时，变量默认的生存期为自动存储生存期，因此经常会省掉关键词。
- 静态存储生存期。由关键字 **static** 声明的变量是静态变量，它在整个程序的执行期间会一直存在。静态变量可能仍然是一个块内的局部变量，但是可在块外存在。如果一个变量在所有块外被声明，那么不管声明时是否使用 **static** 关键词，它均会被指定为静态存储。

17.6.4 连接

不管变量是由一个文件使用或者由多个文件使用，考虑如何识别这些相同的变量

名并将它们指向同一个变量是非常重要的。下面是 3 种不同类型的连接。

- 外部连接。如果一个变量或者函数在外部被连接，那么能够在整个程序中所有声明它的地方来使用。变量名是由连接器来识别的。如果使用 **extern** 关键词来声明或者在所有块外声明并且声明时不指定存储类型，那么变量就有一个外部连接。
- 内部连接。如果在所有函数之外被声明为 **static** 类型，变量将有一个内部连接。对于翻译单元来说，这个变量仍然为内部的，但是它能在整个程序中被识别。连接器并不“知道”这个变量。
- 无连接。这是所有其他类型变量的状态，比如那些具有自动生存期的变量。

17.6.5 18 系列中指定变量的存储器类型

在 C18 编译器中，可以对变量存放的存储器进行指定，这会增加其复杂度，但从另外一个角度来看，它增加了对变量存放位置的选择。由于 PIC 微控制器是哈佛结构，因此可以灵活地使用存储器，但是这也提出了一个挑战——C 语言如何进行存储分配。为了解决这个问题，C18 编译器引入了变量的存储器修饰关键词 **far** 和 **near**。这 2 个关键字是对 C 语言的有效扩展，并且是 C18 特有的。它们指出了变量占用微控制器存储空间的大小或使用存储器的方式。这 2 个关键词又可以和另外 2 个扩展的关键词 **rom** 和 **ram** 联合起来使用。如果需要指定一个变量或者常数的存储器类型，可以使用这些关键词。如果不指定存储器类型，变量默认为 **ram** 和 **far** 类型的。表 A6-8 总结了所有这些存储器修饰关键词并给出了简短的描述。

在程序中还可以指定存储器模型：*small* 和 *large*。表 A6-9 总结了这 2 个存储器模型的含义。存储器模型是通过命令行参数来指定的，默认为 small 类型。这 2 种模型的唯一区别是指针的大小。如果程序规模较小或者中等大小，使用默认的 small 模型即可。

17.6.6 存储类型举例

在 **c018i.c** 文件的前两段代码的开始处，我们看到了一些变量的声明。在这里，显式地使用了关键词 **extern** 来指示变量有一个外部连接。这 3 个变量在 **18f242.h** 文件中也出现了。存储器修饰关键词 **near** 指示变量是存放在可读写的 RAM 中的。有一些数据类型是以前我们未遇到过的，可以参考表 A6-4 的描述。 455

```
extern volatile near unsigned long short TBLPTR;
extern near unsigned FSR0;
extern near char FPFLAGS;
```

下面的第 2 个例子是截取 **_do_cinit()** 函数的开始部分（去掉了注释部分）。在函数开始处声明了 4 个变量。每个变量都是静态的，所以它们在程序运行期间都将一直存在。它们声明在一个块（函数体）内。对于这个函数来说，它们是局部变量。

```
void _do_cinit (void)
{
  static short long prom;
  static unsigned short curr_byte;
  static unsigned short curr_entry;
  static short long data_ptr;
...
```

在 17.7.3 节开始仿真 **c018i.c** 文件时，会继续深入地研究这些变量声明方式的含义。

17.7 启动文件：c018i.c

最后，我们来完整地学习一下 **c018i.c** 文件。很奇怪的是，在仿真第一个 C 程序时，仿真器没有直接运行到 **main** 函数处。的确，所有教科书都告诉我们程序应该从 **main** 函数开始执行，难道 **main** 函数不是从复位向量处启动吗？其实，为了保证 C 程序的正常运行，必须在 **main** 函数开始运行之前初始化一些环境变量。这些初始化包括 C 程序正确运行所需要的任何初始化操作，比如初始化用于数据传输的软件栈或者所有变量和常数。这些初始化操作可能是由 C 语言本身所要求的，或者是由于程序中的变量在运行之前需要初始化。每一个编译器都包含执行这些初始化操作的例程，对于程序员来说，它们基本是不可见的。但是，如果我们依赖于其中的某些初始化操作，还是有必要大致了解一下这些用于初始化的例程。

17.7.1 C18 启动文件

C18 编译器提供了 3 个不同复杂度的启动文件。它们是已编译过的目标文件，并在构建阶段被连接到主程序。通常在连接器脚本中设置启动文件，然后连接器将它连接到用户的应用程序。你可以获得这些启动文件的源代码。由于启动程序是放置在复位向量处的，因此它是 CPU 上电复位后执行的第一个程序。它的作用是初始化软件栈以及一些具有初始值的变量。初始化完毕，程序跳转到用户的 **main** 函数处开始执行。

456

到现在为止，我们讲解的例程都是利用 **c018i.c** 这个启动文件。当使用 MPSIM™ 仿真任何一个 C 例程时，如果一开始就单步执行程序，你会看到这个启动文件进行的初始化操作。这个文件是为那些运行在非扩展模式的处理器上的程序编写的启动文件。而 **c018i_e.c** 版本的启动文件则是运行在扩展模式的处理器上的。

启动文件的一个简单版本是 **c018.c** 文件。它仅仅设置软件栈，之后程序就跳转到 **main** 函数处，没有对任何与存储器相关数据进行初始化。一个较为复杂的版本是 **c018iz.c** 文件。它执行的操作和 **c018i.c** 相同，但是由于与严格的 ANSI C 要求一致，它还会将未初始化的变量全部初始化为 0。这两个版本的启动文件是用于非扩展模式的操作。它们用于扩展模式操作的等价启动文件有 **c018_e.c** 和 **c018iz_e.c** 文件。

17.7.2 c018i.c 文件的结构

在 17.3.1 节我们已经引用了 **c018i.c** 这个启动文件的开始部分。

启动文件的第 1 个操作是初始化 **FSR1** 和 **FSR2**，这两个寄存器用于软件栈(因此，程序员不能访问它们)。然后程序调用函数 **_do_cinit**。这个函数将初始化那些声明在 RAM 中的变量(如果有的话)。在这个函数的结尾处，程序调用 **main** 函数，如下几行所示。

```
loop:
  // Call the user's main routine
  main ();
  goto loop;
```

在这里我们可以发现一个有趣的现象：如果 **main** 函数执行一个 return 语句，之后会立即再次调用 **main** 函数。

17.7.3 仿真 c018i.c 文件

在仿真本书程序时，我们已经多次跳过 **c018i.c** 这个启动文件。但是，现在我们进入这个程序查看一下它进行了什么操作。我们也可以通过这个程序来检验一下学过的例子中一些特定变量的存储类型。

打开你为例程 14-1 创建的项目，并启用 MPLAB 仿真器。打开 watch 窗口，然后选择观察如图 17-3 所示的变量。之后，复位仿真器。我们发现了一个有趣的现象：同声明在程序开始处的 **TBLPTR** 变量一样，在源文件中声明的变量显示为有效。Watch 窗口中的其他变量(声明在 **_do_cinit** 函数中)指示为"out of scope"，如图所示。这同 17.6.2 节中所描述的一致。单步执行程序直到进入 **_do_cinit** 函数。注意一下这 3 个变量是如何突然进入可见范围，并获得一个(0)值的。如果继续单步执行函数，你会发现程序从该函数返回并执行到调用处。由于程序简单，它不需要初始化。然后程序调用 **main** 函数。

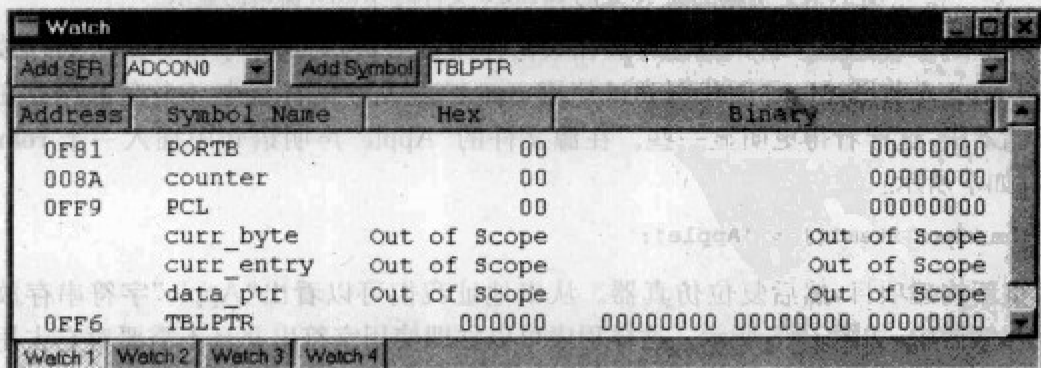


图 17-3 仿真例程 14-1 的 **c018i.o** 文件时 Watch 窗口的显示

现在,在源代码中将 **counter** 的声明语句移动到 **main** 函数中。**counter** 的声明在源代码如下所示的行中:

457

```
unsigned char counter;    //specify counter as unsigned character
```

重新构建项目,然后仿真程序。现在注意到在程序开始执行时,**counter** 指示为"out of scope"。由于声明在函数内部,它失去了外部连接,如表 17-2 所示。如果继续单步执行程序,你会发现在程序执行到 **main** 函数时,**counter** 变为可见。

现在打开例程 16-2 的项目,这个程序中有大量的数组需要初始化。打开一个 watch 窗口,并观察如图 17-4 所示变量。窗口最底下的 3 个变量在程序开始时显示超出了可见范围。单步执行程序直到进入 **_do_cinit** 函数。像以前一样,此时下面的 3 个变量获得一个数值。继续单步执行,注意到程序运行到该函数的一个主循环体内。最后,你会发现作为数据的字符串从程序存储器移动到数据存储器。图中显示"Apple"字符串的一部分完成了移动。

Watch

Add SFR | ADCON0 | Add Symbol | _tmp_0

Address	Symbol Name	Hex	Binary	Char
0080	⊞ list			
00A3	counter	00	00000000	.
0088	number	00	00000000	.
0089	⊟ item1			
0089	[0]	41	01000001	A
008A	[1]	70	01110000	p
008B	[2]	70	01110000	p
008C	[3]	00	00000000	.
008D	[4]	00	00000000	.
008E	[5]	00	00000000	.
008F	⊞ item2			
009C	curr_byte	000D	00000000 00001101	..
009E	curr_entry	0001	00000000 00000001	..
00A0	data_ptr	000038	00000000 00000000 00111000	..8

Watch 1 | Watch 2 | Watch 3 | Watch 4

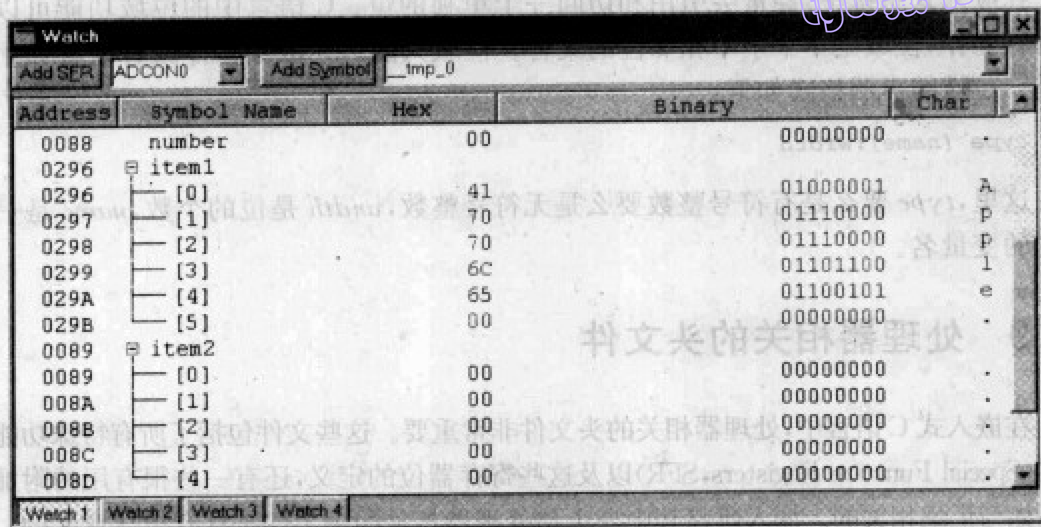
458

图 17-4 仿真例程 16-2 的 **c018i.o** 文件时 Watch 窗口的显示

我们刚才看到的字符串只是存放在数据存储器中,因为这是字符串默认的存放位置,如 17.6.5 节所述。下面我们尝试使用 **rom** 扩展关键词将其中一个变量存放在程序存储器中,这样看得更明显一些。在源文件的"Apple"声明语句中插入一个 **rom** 关键词,如下所示:

```
rom char item1[] = "Apple";
```

重新构建项目,然后复位仿真器。从串地址我们可以看出"Apple"字符串存放到程序存储器中,如图 17-5 所示。这样程序可以立即使用字符串并且不需要对它占用的内存进行初始化。



Address	Symbol Name	Hex	Binary	Char
0088	number	00	00000000	.
0296	item1			
0296	[0]	41	01000001	A
0297	[1]	70	01110000	p
0298	[2]	70	01110000	p
0299	[3]	6C	01101100	l
029A	[4]	65	01100101	e
029B	[5]	00	00000000	.
0089	item2			
0089	[0]	00	00000000	.
008A	[1]	00	00000000	.
008B	[2]	00	00000000	.
008C	[3]	00	00000000	.
008D	[4]	00	00000000	.

图 17-5 仿真例程 16-2——使用 **rom** 关键字时 Watch 窗口的显示

17.8 结构体、联合体和位域

在下一节，我们将学习微控制器的头文件。由于大部分的头文件应用了一些我们至今还未遇到过的数据类型，因此我们先对它们进行介绍。

结构体和联合体都是一组有关联的变量的集合，它们分别使用 C 语言的关键字 **struct** 和 **union** 来定义。从某个角度来看，它们像是一个数组，只是可以包含不同类型的数据元素。

结构体中的元素称为成员，在存储器中按先后顺序排列——成员占据相邻的存储器单元。声明一个结构体时，使用 **struct** 关键词，后面可以给出一个可选结构体名，接着是结构体成员列表，要对每个成员自身进行声明。例如：

```
struct resistor {int val; char pow; char tol;};
```

声明了一个 **resistor** 结构体。它包含电阻的值(**val**)、额定功率(**pow**)和误差(**tol**)。结构体名可以放在大括号之前或之后。

结构体中的元素可以通过指定结构体变量名和结构体成员名，中间再加上英文句号的形式来引用。因此，**resistor. val** 引用了上面的示例结构体的第一个成员。

与结构体一样，联合体也能包含不同类型的变量。与结构体不同的是，联合体中元素的起始地址都一样。因此，在一个时刻联合体只能代表其中一个成员，并且联合体的大小为其中最大元素的大小。这就需要由程序员在程序中跟踪当前联合体中存放的是何种类型的元素。声明联合体的方法与声明结构体的方法类似。

联合体、结构体和数组可以结合起来使用。在后续部分中，我们会学习这种使用方式的一个例子。

在嵌入式系统中，经常会引用和访问一个单独的位。C语言中的位域功能可以完成这种操作，位域是单个字中相邻位的集合。位域只能声明成结构体或者联合体中的成员。位域的声明格式如下：

```
type [name]:width
```

这里，*type* 要么是有符号整数要么是无符号整数，*width* 是位的个数，*name* 是一个可选的变量名。

17.9 处理器相关的头文件

在嵌入式C语言中，处理器相关的头文件非常重要。这些文件包括了所有特殊功能寄存器(Special Function Registers，SFR)以及这些寄存器位的定义，还有一些很有用的附加定义例如与汇编相关的。深入研究其中一个与处理器相关的头文件是具有指导意义的。我们将查看 **18f242.h** 这个头文件。你可以在C18软件目录 **mcc18\h** 下找到这个文件。

17.9.1 SFR 定义

例程17-4是18F242头文件的摘录。在这里，定义了端口B和它的位。摘录中的第一行使用了不少于4个的C语言关键字来定义 **PORTB** 变量。使用 **unsigned char** 将它声明为一个单字节，同时使用 **volatile** 关键字指出它可以在程序流程之外被修改。最后，**extern** 关键字指出变量可以在这个文件之外被访问。**near** 关键字指出它存放在可读写RAM中。

460

仔细查看例程，可以发现端口位组合在一起并被声明成 **PORTBbits** 联合体，其中又包括了2个结构体。同端口B的声明一样，这个联合体指定为 **extern volatile near** 类型。联合体中的第1个结构体是端口位的常规命名列表，每个端口位声明为单比特的位域。

第2个结构体是端口位另外一种命名方式的列表，每个端口位仍然为单比特的位域。由于这2个结构体同属于一个联合体，它们实际上占用相同的存储空间，可以选择使用其中一个结构体。

现在我们至少能理解前面几章对端口和其他SFR位的引用。比如，当写

```
PORTBbits.RB7 = 1;
```

语句时，我们现在知道它是在引用结构体成员 **RB7**（位域变量），而这个结构体又是联合体 **PORTBbits** 的一个成员。

例程17-4　端口B的声明——18F242.h文件(版本1.6)部分内容

```
extern volatile near unsigned char PORTB;
extern volatile near union {
  struct {
    unsigned RB0:1;
    unsigned RB1:1;
```

```
        unsigned RB2:1;
        unsigned RB3:1;
        unsigned RB4:1;
        unsigned RB5:1;
        unsigned RB6:1;
        unsigned RB7:1;
    };
    struct {
        unsigned INT0:1;
        unsigned INT1:1;
        unsigned INT2:1;
        unsigned CCP2:1;
        unsigned :1;
        unsigned PGM:1;
        unsigned PGC:1;
        unsigned PGD:1;
    };
} PORTBbits;
```

17.9.2 头文件中汇编相关的定义

例程 17-5 显示了头文件中使用了 **#define** 预定义来定义 PIC 18 系列微控制器特定的汇编指令。通过这种方法，可以在 C 程序中使用这些函数，而不再需要使用内嵌汇编的方法。如果不经意地看，它们似乎以函数形式呈现。在第 19 章讲 Salvo 实时操作系统时，将再次使用这种方法。 461

例程 17-5 18F242.h 文件部分内容——汇编基本程序

```
...
#define Nop()     {_asm nop _endasm}
#define ClrWdt()  {_asm clrwdt _endasm}
#define Sleep()   {_asm sleep _endasm}
#define Reset()   {_asm reset _endasm}
...
```

17.10 深入学习——MPLAB 连接器和.map 文件

图 14-1 显示出连接器在任何一个构建过程中扮演的重要作用。对于简单的应用来说，程序员并不需要知道连接器是如何工作的。但是即使是一个简单的应用，大概了解连接器也可以帮助我们理解程序是如何组装起来的——特别是连接器如何寻找和分析需要连接的那些“隐藏”文件。对于更复杂的应用，程序员可能需要修改或者重写已提供的连接器脚本。这一节我们来介绍 MPLAB 连接器 MPLINK™。参考文献 17.2 给出了这方面的一些相关信息。

17.10.1 连接器的功能

如 14-1 图所示的程序构建过程中，连接器以目标文件作为输入，然后将这些目标

文件组装在一起产生一个能下载到微控制器上运行的可执行代码。连接器也能够产生一些辅助调试的信息，如存储空间的分配。目标文件可能是由C语言或者汇编语言编写的应用代码，也可能是通用的库文件。不管是哪种，它们包含的代码大部分是可重定位的。这意味着还未指定变量的存储器（数据或者程序存储器）地址。而这是连接器需要完成的工作：用来将所有的目标文件定位到存储器中并且保证变量之间能够正确地被引用。连接器也会控制软件栈的分配。这些工作都是在连接器脚本指导下进行的，脚本文件包含目标微控制器存储空间映射的基本信息。连接器在处理这些工作时可以非常容易地发现程序的错误，比如地址冲突或者信息不全。

17.10.2 连接器脚本

连接器脚本是一个文本文件，它由许多的连接器伪指令组成。它的作用是告诉连接器可用的存储空间在哪里以及如何使用这些空间。因此，这些伪指令精确地反映了目标微控制器的存储器资源和映射结构。MPLAB中提供了适用于所有PIC 18系列微控制器的标准连接器脚本。当安装了标准C18软件后，可以在**mcc\lkr**目录下找到它们。**18f242.lkr**是适用于18F242微控制器的连接器脚本，它用于本书中任何一个C语言程序。例程17-6显示出了这个文件的内容。另外还有一个脚本文件和这个很相似，但是它并不局限于某一个特殊的C18实现，你可以在MPLAB的安装目录下找到它。

462

例程 17-6 18F242 的连接器脚本

```
// $Id: 18f242.lkr,v 1.1 2003/12/16 14:53:08 GrosbaJ Exp $
// File: 18f242.lkr
// Sample linker script for the PIC18F242 processor

LIBPATH .

FILES c018i.o
FILES clib.lib
FILES p18f242.lib

CODEPAGE    NAME=vectors   START=0x0        END=0x29       PROTECTED
CODEPAGE    NAME=page      START=0x2A       END=0x3FFF
CODEPAGE    NAME=idlocs    START=0x200000   END=0x200007   PROTECTED
CODEPAGE    NAME=config    START=0x300000   END=0x30000D   PROTECTED
CODEPAGE    NAME=devid     START=0x3FFFFE   END=0x3FFFFF   PROTECTED
CODEPAGE    NAME=eedata    START=0xF00000   END=0xF000FF   PROTECTED

ACCESSBANK  NAME=accessram START=0x0        END=0x7F
DATABANK    NAME=gpr0      START=0x80       END=0xFF
DATABANK    NAME=gpr1      START=0x100      END=0x1FF
DATABANK    NAME=gpr2      START=0x200      END=0x2FF
ACCESSBANK  NAME=accesssfr START=0xF80      END=0xFFF      PROTECTED

SECTION     NAME=CONFIG    ROM=config

STACK SIZE=0x100 RAM=gpr2
```

现在我们来学习这个连接器脚本文件示例。我们的目的是理解它所表达的含义，而不是重新写一个连接器脚本。因此，我们不用考虑它们严格的语法格式。

- 连接器脚本的注释。所有的注释均以//开头。连接器忽略其后跟随的一行文本。正如通常的做法，这里的注释给出了文件的标题以及版本信息。
- **LIBPATH** 伪指令。它提供一个可选的用于搜索包含文件的目录。在这个例子中没有使用它。
- **FILES** 伪指令。这个伪指令指定需要连接进来的目标文件。这里指定了 3 个文件。
  - —**c018i.0**。这是启动文件的目标代码，它在本章前面已经描述过。
  - —**clib.lib**。它是 C18 编译器支持的标准 C 函数库。
  - —**p18f242.lib**。这个库文件包含与处理器相关的信息以及与可提高编程效率的处理器相关的头文件。
- **CODEPAGE** 伪指令。这个伪指令用来分配程序存储器。在这个例子中，使用它的超过了 6 次，它的主要作用是描述微控制器中存储器的映射结构。连接器可分配给程序代码的主要存储空间是从地址 $02A_H$～$03FF_H$。这同图 12-6 中描述的存储器映射结构是一致的。其上的空间是为中断向量表保留的。配置数据和设备标识所占用的存储器块也是保留的，对应于表 12-4 中相应的存储地址。其他还有一些存储空间是为 EEPROM 和其标示保留的。
- **ACCESSBANK** 伪指令。例子中 2 次使用到这个伪指令，它用于分配可读写的数据存储器。第 1 次使用时，定位到可读写存储器中的 RAM 被命名为 **accessram**，地址范围从 0～$7F_H$。第 2 次使用时，它确定了 SFR 存储器块。该存储器块被命名为 **accessfr**，并定位到存储器映射结构中。并且这个存储器块被指定为受保护类型，这样会阻止连接器将其分配为通用空间。因此，在其他地方对 SFR 寄存器进行绝对存储分配是不允许的。
- **DATABANK** 伪指令。这个伪指令类似于 **ACCESSBANK** 伪指令，且它们的使用格式是相同的。它用来指定划分了存储区的 RAM。在这个例子中，可以发现它的实现是完全依据图 12-4 中的数据存储器映射图。连接器可以利用其中每一块，所以没有任何块是受保护的。
- **SECTION** 伪指令。这个伪指令将一个在源代码中使用 **#pragma** 伪指令指定的名称连接到由连接器脚本确定的存储块中。因此，它就为配置空间建立了联系，使得由 **#pragma config**(见例程 17-2)产生的数据可以被放置到正确的存储器地址。
- **STACK SIZE** 伪指令。这个伪指令用于指定软件栈的地址和大小。在这个例子中，它指定了一个大小为 100_H、地址在 RAM 第 2 个区的软件栈。

17.10.3 .map 文件

程序通过连接之后，所有的代码都被正确地映射到不同的存储空间。我们如何来

检验这个论断呢？答案在. map 文件中，它是一个可选文件，可要求连接器产生它。这个文件显示了所有的存储器分配。对于给定的项目，可以单击菜单 **Project > Build Options > Project > MPLINK Linker > Generate map File** 来产生。

当成功构建一个项目之后，. map 文件随同其他输出文件一起被产生，其文件名为项目名称. map。. map 文件内容较多，包括所有符号的存储器地址以及由连接器脚本获得的存储器的映射结构，因此不太好查看。但是，如果程序的运行出现问题，我们需要了解一个变量是如何被处理的，或者一个存储器单元或存储块发生了什么动作，可以用这个文件进行调试诊断。. map 文件的另外一个有用的特性是它显示出存储器的使用比例。当然，随着程序尺寸的增长，这个功能将变得更加重要。

例程 17-7 显示了例程 14-1 程序的. map 文件片断。它显示出一些主程序段被放置的位置以及程序存储器 1%的使用率！

例程 17-7 例程 14-1 的. map 文件片断

```
                            Section Info
           Section         Type      Address    Location  Size(Bytes)
         ---------    ---------    ---------   ---------   ---------
       _entry_scn         code     0x000000     program    0x000006
           .cinit      romdata     0x00002a     program    0x000002
       _cinit_scn         code     0x00002c     program    0x00009e
  .code_example1.o        code     0x0000ca     program    0x000024
     _startup_scn         code     0x0000ee     program    0x000018
...

                       Program Memory Usage
                          Start          End
                      ---------    ---------
                       0x000000     0x000005
                       0x00002a     0x000105
  226 out of 16664 program addresses used, program memory utilization is 1%
```

464

小结

- ☐ 有时候，程序员需要跳出 C 语言严格的规范，而直接使用汇编语言编程。
- ☐ 在 C 语言中使用汇编语言并不难，但是需要理解如何定义中断向量以及如何构造中断服务程序。
- ☐ 编写较大的程序时，理解一些不同的数据类型和变量的存储类型是有益的。
- ☐ 进行 C 语言程序开发远不止于编写源代码。程序员还需要选择（和必须）使用其他更多的文件。还要理解这些文件有哪些、它们的作用以及它们之间如何相互联系，这对程序开发是非常有用的。
- ☐ 连接器将所有需要连接的文件组装在一起。粗浅地理解连接器的功能对于简单的程序是有益的。当编写重要的软件模块时，详细了解连接器是非常重要的。

在这 4 章对 C 语言的讲述之后，我们对 C 语言如何应用到嵌入式系统这方面有了一个初步但很有用的了解。这就可以继续进行一些更复杂的 C 程序的编写，在第 19 章还将继续这个话题。尽管介绍了 C 语言的基本知识，但还是有很多 C 语言知识没有讲述。更深入和更多的 C 语言知识可以通过多实践、学习典型例程、阅读专门的 C 语言书籍来掌握。

参考文献

17.1. PIC18 Configuration Settings Addendum (2005). Microchip Technology Inc., DS51537C.

17.2. MPASM™ Assembler, MPLINK™ Object Linker, MPLIB™ Object Librarian User's Guide (2005). Microchip Technology Inc., DS33014J ; www.microchip.com

第 18 章
多任务实时操作系统

几乎每个嵌入式系统都不止完成一个任务。比如运行在 AGV 上的程序通过碰撞和光学传感器来感知外部环境，测量 AGV 行驶的距离，依据前面的测量结果来计算并实施电机驱动。当系统变得更复杂时，平衡系统中不同的任务就会变得更难。每个任务都将竞相使用 CPU，因此会导致系统其他部分的延迟执行。需要设计一种方法来对程序中存在的不同活动的运行时间进行"公平"地划分。

与分时共享 CPU 同等重要的另外一个问题是确保程序活动能够及时发生。这在几乎所有的嵌入式系统中都非常重要，当系统中存在多个程序活动竞相使用 CPU 时，问题将会变得更加严重。

本章将分析系统在执行多个程序活动时的需求，并仔细研究存在的基本挑战并提出解决它们的策略——实时操作系统。它将导致一个全新的程序设计——不再由程序次序来确定下一步的执行动作，而是由操作系统来控制。

在本章你将学到：

❑ 由多任务引起的挑战；

❑ "实时"的含义；

❑ 如何通过顺序编程来实现一个简单的多任务；

❑ 实时操作系统的基本原理。

由于本章只是概括性地介绍实时编程的概念，因此没有包括实际的程序例子。但是本章是第 19 章的铺垫，在下一章中你将学到许多重要的例子。

18.1 由多任务和实时引发的挑战

在这个忙碌的现代社会中，很多人都在同时完成多个任务。父亲可能需要为 2 个或者 3 个孩子上学做准备——一个孩子的袜子找不到了，一个孩子生病了，另外一个孩子把牛奶打翻了。而且同时狗等着喂食、炖锅正在沸腾、邮递员刚好在敲门、电话也在响。有许多事情需要做，但是在同一个时刻我们只能做一件事情。嵌入式系统中的微控制器同样也有着类似的烦恼。它也会被许多事情缠身，每一个都需要它来处理。它需要决定先处理哪件事情，哪些事情可以稍后再处理。

在本章，我们先来研究多任务和实时的本质。

18.1.1 多任务——任务、优先权、截止时间

图 18-1 是第 16 章中讲到的 AGV 寻光程序的简化流程图。在那里我们主要用这个程序来介绍 C 语言中的特定概念，没有花时间来考虑整个程序的结构。

程序由许多不同的活动组成，每一个活动均出现在流程图中。我们直接采纳这种活动的惯例称呼——任务。任务是一个具有明确目的和结果的程序段。而多任务只是描述了有多个任务需要同时执行的情形。这个例子划分出了 4 个任务。

在程序中，任务被放在一个超循环中，程序依次轮流执行这些任务。程序每循环 10 次，显示任务（将数据传送到手动控制器的 LCD 上进行显示）才执行一次。一次循环以延时任务结束，它确定了循环重复的频率，因此也就影响了程序的时间特性。



图 18-1 AGV 寻光程序的简化流程图

这是一个简单的多任务例程。程序严格地以轮转方式执行其中包含的任务，并且在开始新一轮循环之前先空闲一段时间。

当然，在实际中不同任务的紧迫性不一样。再以那个苦恼的父亲为例：如果电话响了，那么狗很可能必须等一下才能吃东西。因此，我们意识到不同的任务具有不同的优先级。高优先级的任务应该在低优先级的任务开始之前被执行。每个任务也有一个截止时间或者估算的截止时间。例如，电话必须在 30 秒内接听，否则电话会被挂断；孩子们必须在早上 8 点半左右准备离开，否则他们会错过公车，等等。优先级的概念与截止时间的概念有了联系。一个临近截止时间的任务可设置一个高的优先

级——而截止时间靠后的任务的优先级会低一些。

表 18-1 显示了这个例程中的任务以及它们初步的优先级分类。使用了 3 种优先级和 3 种估算的截止时间。在分配优先级时，微动开关被按作为一个紧急事件——这说明 AGV 已经碰撞到物体，电机可能已经停止运转。因此，它被设置成高优先级。相反，使用者很少会注意到显示函数延迟了 1 秒左右。因此，它被设置成低优先级。

表 18-1　AGV 寻光程序中的任务

任　务	优先级	截止时刻(ms)
对微动开关做出反应	1	20
读 LDR	2	50
计算和设置电机速度	2	50
显示	3	500

18.1.2　"实时"的含义

AGV 寻光程序(或者那个苦恼的父亲)是如何与实时这个概念联系起来的？实时的概念虽然被广泛讨论，但还是很模糊。

"实时操作"的一个简单但是完全有效的定义(已经在参考文献 1.1 中被采纳)如下：

实时的系统操作必须能够在给定的截止时间内提供正确的结果。

注意，这个定义并没有说实时运行就意味着速度快，尽管它通常会提供帮助。它只是简单地陈述：需要的结果必须在给定的时间内准备好。因此，如果父亲能够使所有的孩子都能按时上学、在狗哀号之前给它喂食、在邮递员离开之前打开门、在电话挂断之前接听电话，那么他就满足了这个环境下对实时的要求。类似地，如果 AGV 上的程序满足了表 18-1 给出的所有截止时间，那么它的操作也是实时的。

这个简单的定义以及它表达的所有含义都引发出很多程序设计方面的挑战。本章后续部分将介绍这些挑战以及解决它们的方法。

468

18.2　通过顺序编程来实现多任务

到现在为止，我们从事的程序设计，不论是使用汇编语言还是 C 语言编写——有时都被称为*顺序编程*。它直观地表达了程序以正常的方式执行：一个指令或者语句接着上一个依次执行，除非遇到程序分支或者子例程或者函数调用。本章后续部分将不再使用这种类型的程序。从现在开始，我们来研究如何优化顺序编程来使它适合多任务的应用，并找出图 18-1 中程序结构的缺点。

18.2.1 分析超循环

图 18-1 中的这个程序似乎可以很正常地运行，但这主要归因于我们对程序的要求不太苛刻。让我们来分析一下这个程序的一些缺点。这些相互之间存在关联的缺点如下所述。

- 每次循环执行的时间不是常数。一次循环执行的时间等于每个任务的执行时间加上延迟的时间。很明显，这个时间会发生变化。比如，如果在某一次循环中，微动开关被激活，AGV 倒退，然后转弯。这将导致一次执行时间特别长的循环，从而会干扰循环内的其他任务。
- 任务之间相互干扰。一旦一个任务得到机会开始执行，它将会持续占据 CPU 直到任务完成为止。编写任务时，设置了它们的执行时间不能太长。但是，假设显示任务开始执行时，AGV 突然碰到一堵墙。此时，处理器仍然会继续传递数据到显示器，在完成延时程序之后，它才会发现已经发生了一个紧急事件。
- 高优先权的任务没有获得它们所需要的注意。我们已经认识到有些任务比其他任务更加重要。因此，它们应该具有更高的优先级，这个概念已经在中断那一章提到过。而在这个持续的循环中，每一个任务的优先级都是相同的。

我们需要设计一种新方法来构建程序，从而可以识别程序中任务的种类和要求，并且能够满足这些任务实时运行的要求。

18.2.2 时间触发和事件触发的任务

很容易区分时间触发和事件触发的任务。时间触发的任务是在一个特定时间周期结束时发生，通常是周期性的。程序中对光敏电阻的读取就是时间触发的例子。事件触发的任务是在一个特定事件发生时随之发生。按下微动开关就是事件触发的一个很好的例子。

18.2.3 使用中断来区分优先级——前台/后台结构

针对 AGV 寻光程序中任务的优先级问题，一个可行的解决方法是使用中断来实现高优先级的任务。于是这些高优先级的任务在需要时就能立即占用 CPU，特别是当只使用一个中断时。那么程序的结构将如图 18-2 所示。

循环中的任务很可能是时间触发的，那么它们可以利用循环的重复频率作为时基。中断驱动的任务很可能是事件触发的。

通过这个简单的程序结构，我们实现了稳定的循环重复频率和对任务优先级进行区分。这个结构有时被称为前台/后台程序结构。具有较高优先级并由中断驱动的任务处于前台(当它们需要时)，循环中具有较低优先级的任务几乎可以连续地在后台运行。

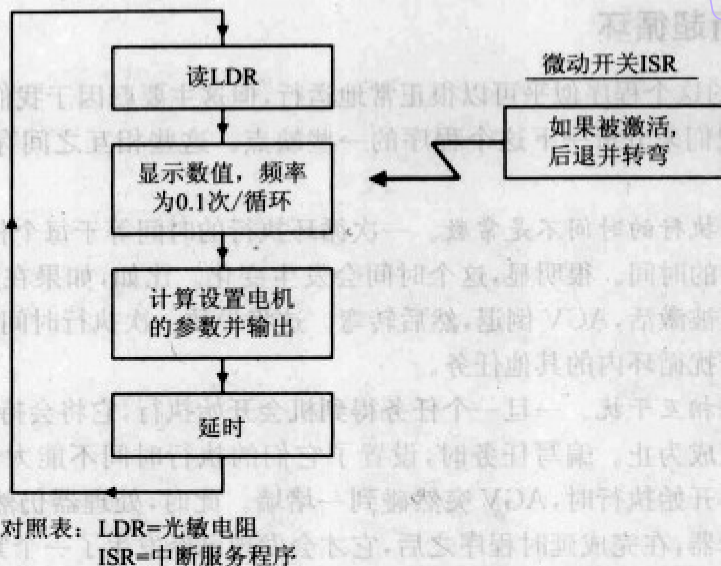


图 18-2 在 AGV 寻光程序中使用中断来区分优先级

18.2.4 引入“时钟滴答”来同步程序活动

为了减少执行时间可变的任务对总的循环执行时间的影响，可以设置一个定时中断来触发整个循环的执行，比如使用定时器的溢出中断。那么程序的结构将如图 18-3 所示。现在主循环由定时器的溢出中断来触发，所以循环执行的频率是固定可靠的。时间触发的任务以这个稳定的频率为基础来完成自身的活动。事件触发的任务在需要时通过中断来触发。当然，在编写程序时，任务的执行时间必须进行计算和控制，从而可以使整个循环能在截止时间内完成，并且事件触发的任务不会过多破坏循环时间的重复特性。

这种使用一个常规的定时中断来同步程序活动的方法在第 9 章已经介绍过，并在图 9-3 中进行了说明。在那一章已经提到，它通常被称为“时钟滴答”(clock tick)。这个方法很简单，但是它将成为我们后面许多程序的基础。时钟滴答概念不应该同时钟振荡器混淆，尽管时钟滴答是由振荡器获得的。

470

18.2.5 一个通用的“操作系统”

从图 18-3 中显示的程序结构中我们可以抽象出一个通用的“操作系统”，如图 18-4 所示。在这里，主循环包括一系列低级或者中级优先权的任务，它们由“时钟滴答”驱动。每个任务的一般结构如图中左边所示。每个任务必要时可设置一个使能标志(内存单元中的 1 位)和一个任务计数器。每个“时钟滴答”都需要执行的任务不需要设置标志和计数器。但是，许多任务只需要以较低的频率来执行，可通过设置计数器来设定任务执行的时间间隔。也可以由其他任务或者中断服务程序来设置或者清除任务

的启用标志，从而允许或者禁止任务的运行。

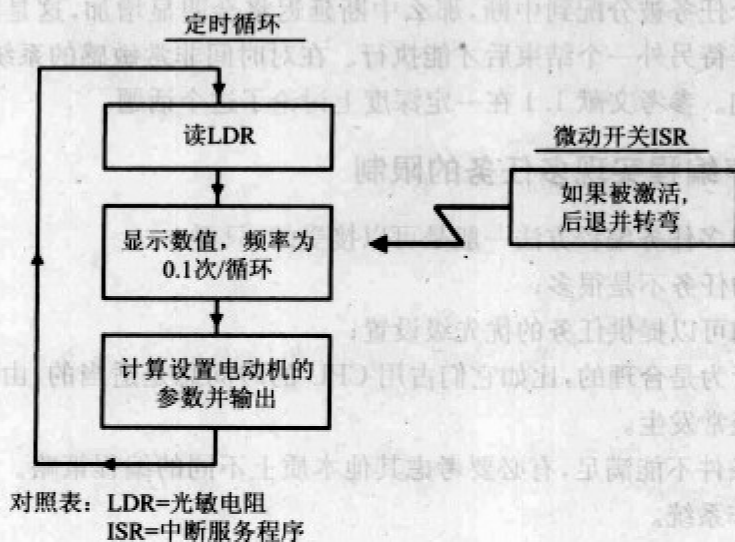


图 18-3 在 AGV 寻光程序中使用定时循环

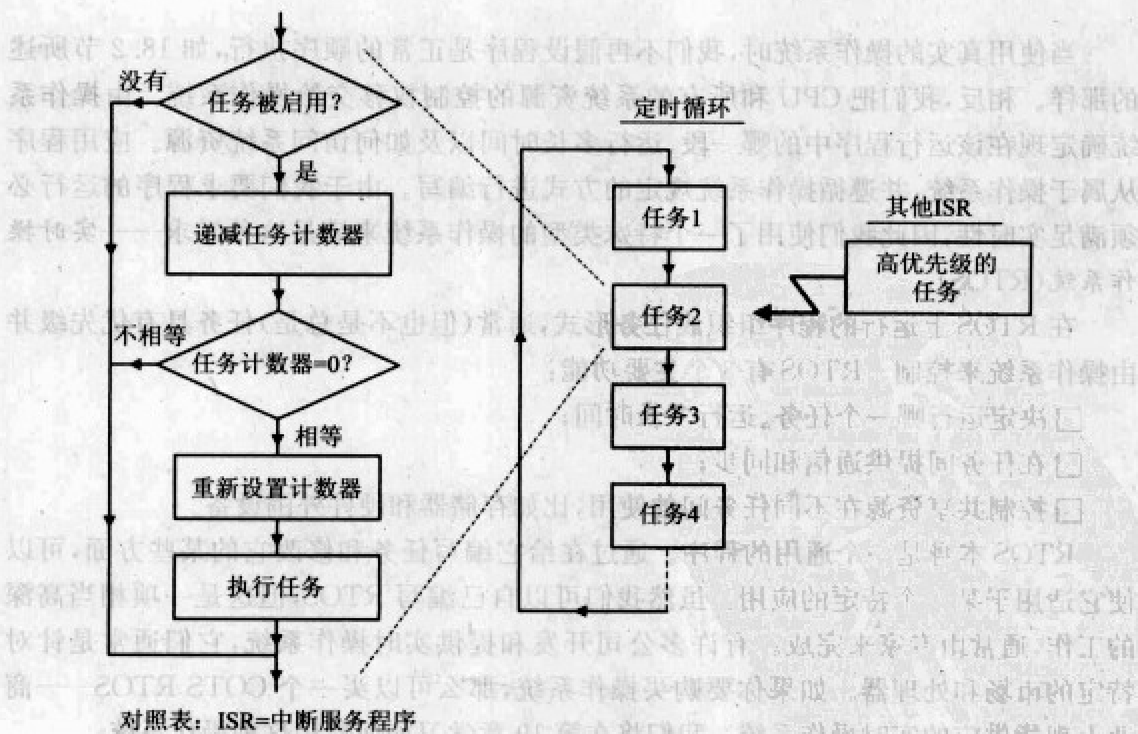


图 18-4 一个通用"操作系统"的结构，使用顺序编程

可以采纳这个通用操作系统的结构来形成一个多任务编程框架。在参考文献18.1和参考文献18.2中将多任务编程框架的一般性概念应用到实际设计中。参考文

献 18.1 描述了一个基于 PIC™ 16F84 微控制器的多任务节拍器的完整设计。

如果有多个任务被分配到中断，那么中断延迟将会明显增加，这是因为一个中断服务程序必须等待另外一个结束后才能执行。在对时间非常敏感的系统中，必须仔细分析延迟的影响。参考文献 1.1 在一定深度上讨论了这个话题。

18.2.6 顺序编程实现多任务的限制

上面描述的多任务编程方法一般是可以接受的，只要：

- ❑ 系统中的任务不是很多；
- ❑ 这种结构可以提供任务的优先级设置；
- ❑ 任务的行为是合理的，比如它们占用 CPU 的时间总是适当的，由中断驱动的任务不是经常发生。

471

如果这些条件不能满足，有必要考虑其他本质上不同的编程策略。一个很好的选择就是实时操作系统。

18.3 实时操作系统

当使用真实的操作系统时，我们不再假设程序是正常的顺序执行，如 18.2 节所述的那样。相反，我们把 CPU 和所有的系统资源的控制权移交给操作系统。由操作系统确定现在该运行程序中的哪一段、运行多长时间以及如何访问系统资源。应用程序从属于操作系统，并遵循操作系统规定的方式进行编写。由于我们要求程序的运行必须满足实时性，因此我们使用了一个特殊类型的操作系统来满足这种需求——实时操作系统(RTOS)。

在 RTOS 上运行的程序组织成任务形式，通常(但也不是总是)任务具有优先级并由操作系统来控制。RTOS 有 3 个主要功能：

- ❑ 决定运行哪一个任务、运行多长时间；
- ❑ 在任务间提供通信和同步；
- ❑ 控制共享资源在不同任务间的使用，比如存储器和硬件外围设备。

472

RTOS 本身是一个通用的程序。通过在给它编写任务和修改它的某些方面，可以使它适用于某一个特定的应用。虽然我们可以自己编写 RTOS，但这是一项相当高深的工作，通常由专家来完成。有许多公司开发和提供实时操作系统，它们通常是针对特定的市场和处理器。如果你要购买操作系统，那么可以买一个 COTS RTOS——商业上现货供应的实时操作系统。我们将在第 19 章学习如何使用这种操作系统。

18.4 调度策略和调度器

RTOS 的一个关键部件是调度器。它确定哪一个任务可以在某一个特定的时刻

运行。除此之外，调度器必须知道哪些任务已准备就绪以及它们的优先级（如果有的话）。有许多本质上不同的调度策略，下面我们来学习一下。

18.4.1 循环调度

循环调度策略比较简单。每一个任务允许在运行完毕之后，再将执行权移交给下一个。任务在运行期间不能被打断。这种调度方式很像本章前面提到的超循环。循环调度策略的一个图表式例子如图 18-5 所示。这里的水平条带代表 CPU 的活动，标注有数字的块代表不同任务的执行。任务轮流被执行。最开始，任务 3 的执行时间最长，任务 2 最短。但是在第 3 次迭代中，任务 1 的时间要长一些，整个循环的时间也就长一些。循环调度的缺点类似于循环中顺序编程的方式，这在前面已经讲述过。但至少这种调度方式非常简单。

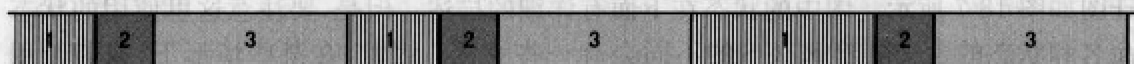


图 18-5 循环调度——任务 1、2 和 3 轮流执行

18.4.2 时间片轮转调度和上下文切换

在时间片轮转调度中，操作系统由一个常规的中断来驱动（即“时钟滴答”）。任务以固定的顺序被选择执行。每一次时钟滴答，当前的任务被打断，下一个任务开始执行。每个任务的优先级相同，它们轮流等待 CPU 时间片。任务不允许一直运行到结束，它能被其他任务抢占，即任务被中途打断。这种调度器属于抢占式调度器。

抢占式任务的上下文切换以及开销是不能忽略的。当任务再次运行时，它必须能够紧接着上一次无缝运行，而不受上次被抢占的影响。因此，在进行上下文切换时，必须保护完整的上下文（所有的标志、寄存器和其他存储单元）。但是，时间紧急的程序活动不应该被中断，这种要求需要在程序中指出。

473

时间片轮转调度策略的一个图表式例子如图 18-6 所示。标注有数字的块还是代表不同任务的执行，但是与图 18-5 中有一个重要的区别。现在，每个任务一次运行的时间是一个固定长度的 CPU 时间片。在图中以箭头表示的时钟滴答将引发任务切换。当时间片结束时，不管当前任务是否完成，下一个任务都将占用 CPU。在图中的某个阶段，任务 2 完成，在后续的几个时间片内不再需要占用 CPU。后来，任务 2 又准

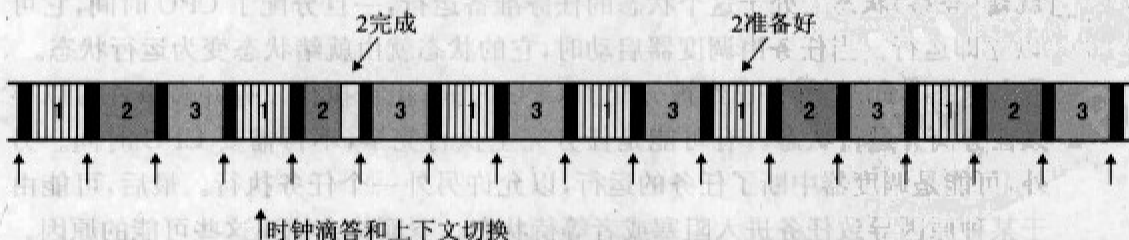


图 18-6 时钟片轮转调度

备运行，在后续周期开始轮流执行。

由于任务和上下文的切换，导致一些不可避免的时间消耗，在图中使用黑色条形块来表示它们。这个时间可以满足 RTOS 的运行需求，但对应用程序来说却是一个时间损失。

18.4.3 任务状态

现在有必要停下来考虑一下在调度器控制下的任务发生了什么事情。很明显，在一个时刻只可能有一个正在运行的任务。其他的任务可能需要运行，但是此时它们不能获得运行的机会。另外还有一些任务的运行需要一个特定的事件集做出反应，因此它只能在程序运行的特定时刻被激活。

因此，认识到任务可以在不同的状态之间变迁是非常重要的。一个可能的状态变迁图如图 18-7 所示。图中的状态在下面有详细的描述。但是，要注意这里使用的状态命名和状态的含义随着 RTOS 的不同会有一些变化。因此，在某些情形下，我们使用了多个名称来描述一个特定的状态。

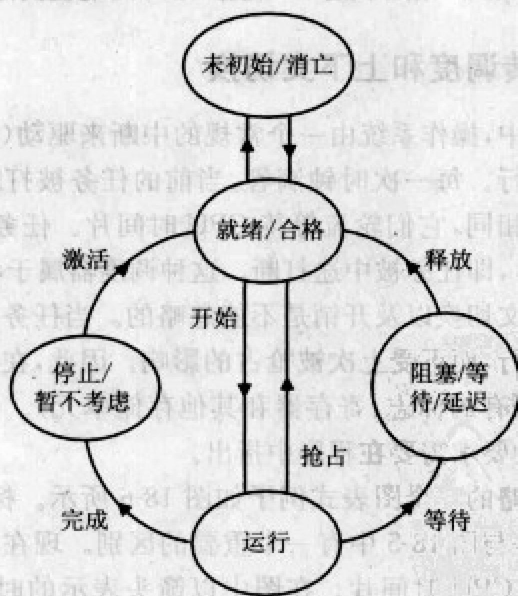


图 18-7 任务状态

- □就绪(合格)状态。处于这个状态的任务准备运行，一旦分配了 CPU 时间，它可以立即运行。当任务由调度器启动时，它的状态就由就绪状态变为运行状态。
- □运行状态。此时，任务已经被分配了 CPU 时间，正在执行。有许多事件可以导致任务离开运行状态。有可能是任务完全执行完毕，不再需要 CPU 时间。另外，可能是调度器中断了任务的运行，以允许另外一个任务执行。最后，可能由于某种原因导致任务进入阻塞或者等待状态。下面将会描述这些可能的原因。
- □阻塞/等待/延迟状态。处于这个状态的任务开始是处于运行状态的，但由于某

种原因不允许继续运行。有许多不同的原因可以导致任务进入这个状态。实际上,如果想了解更多的细节,可以将这个状态分解成多个状态。任务可能正在等待它需要的数据或者等待正被其他任务使用的资源;也有可能需要等待一段时间之后才能继续运行。一旦满足了任务需要的条件,它将离开这个状态。

- 停止/暂不考虑/休眠状态。处于该状态时,任务不再需要 CPU 时间。任务由于任何一种因素可以再次被激活,从而进入就绪状态。
- 未初始/消亡状态。从 RTOS 来看,处于这个状态的任务将不再存在。该状态的含义是在程序执行过程中这个任务不需要持续存在。一般来说,如果程序再次需要该任务,必须重新创建或者初始化。不需要的任务可以先销毁,在需要的时候再创建。从任务列表中清除掉不需要的任务可以简化调度操作,减少内存需求。

18.4.4 抢占式优先级调度

现在我们接着研究前面提到的调度策略,深入理解一下任务的工作方式。正如我们看到的,在时间片轮转调度策略中的任务从属于较高优先级的操作系统。但是除操作系统之外所有其他的任务都具有相同的优先级,因此一个不重要的任务和一个很重要的任务占用 CPU 的机会同样多。如果我们给任务设置不同的优先级,就可以改变这种状况。

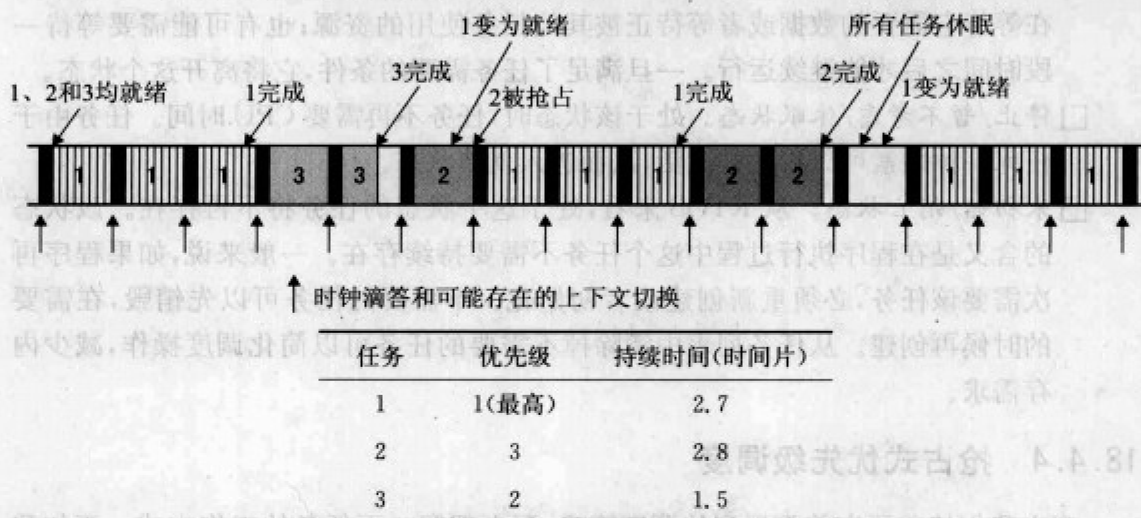
在抢占式优先级调度器中,每个任务可以具有不同优先级。现在,高优先级的任务允许在低优先级的任务之前完成,而不管低优先级的任务完成何种功能。调度器仍然在时钟滴答下工作。每一次时钟滴答,调度器开始检查已经就绪的具有最高优先级的任务。优先级最高的任务将获得 CPU 的使用权。此时,如果有一个正在运行的任务仍然需要运行,并且它的优先级最高,那么它将继续占用 CPU。如果是一个低优先级的任务正在运行,它将被一个更高优先级的任务取代,并且会由运行状态变为就绪状态。因此,高优先级的任务变成"运动场上欺凌弱小者"。几乎在任何场合下它都可以首先被执行。

475

图 18-8 说明了这种调度策略的工作方式。这个例子包括许多 RTOS 的关键概念,值得认真理解。图中有 3 个任务,每个任务都具有不同的优先级和执行时间。开始,所有的任务都处于就绪状态。由于任务 1 具有最高的优先级,因此调度器选择它运行。在下一个时钟滴答,调度器发现任务 1 仍然需要运行,所以允许它继续运行。下一个时钟滴答仍然是这种情况,任务 1 在下一个时间片完成。现在任务 1 暂时不需要 CPU 时间,它被暂不考虑。因此,在下一个时钟滴答调度器选择具有最高优先级且已就绪的任务,此时就是任务 3。它也会一直运行直到结束。

终于,任务 2 获得机会运行。但是,很不幸,在它运行的第一个时间片期间任务 1 重新变为就绪状态。因此,在下一个时钟滴答,调度器再次选择任务 1 运行。它又被允许直接运行到结束。当且仅当没有其他任务处于就绪状态时,任务 2 才重新开始执

行，最终完成。接下来的一个时间片，由于没有任何活动的任务，因此 CPU 空闲。之后，任务 1 再一次就绪，又开始运行直到结束。



任务	优先级	持续时间(时间片)
1	1(最高)	2.7
2	3	2.8
3	2	1.5

图 18-8 抢占式优先级调度

18.4.5 协作式调度

刚才描述的抢占式优先级调度策略是一种典型的 RTOS 调度策略。但是它也并不是毫无缺点。调度器必须为每一个被抢占的任务进行完整的上下文保护。这些现场信息通常保存在每个任务的栈中，这将非常耗费存储空间。上下文切换同样也很费时间。进一步，任务必须以一种方式来编写，该方式可以保证任务在运行的任何时刻均可以被切换。

476

另外一种抢先调度策略是协作式调度。现在，每一个任务都可以根据自身运行情况来选择放弃 CPU 的时刻。这看起来像是我们在放弃使用操作系统的调度功能，但是如果每个任务都能够正确编写，就不会出现这种情况。这种方式的优点是由于任务可以选择何时放弃 CPU，于是它可以控制上下文保护的时机，从而可以减少关键的任务切换开销。

协作式调度不可能像抢先式调度那样能够对临近截止时刻的任务做出同样快的反应。但是它需要更少的存储空间，而且任务切换也要快一些。这在小系统中是非常重要的，比如一个基于 PIC 微控制器的系统。

18.4.6 中断在任务调度中的作用

到现在为止，我们还未提到中断与 RTOS 的联系。是否可以使用中断服务程序来实现一个任务，就像我们在图 18-4 中的那种做法？答案是否定的。中断几乎总是首先用于提供时钟滴答——通过一个定时器的溢出中断来产生。除此之外，中断服务程序通常用于提供一些紧急的信息给任务或者调度器。比如，中断可以配置成一个特定事

件发生的信号，如果信号被触发，那么会将任务从阻塞状态释放（图18-7）。因此，正常情况下，中断服务程序本身不作为一个任务。

18.5 任务开发

前面我们已经建立了任务的概念以及如何调度它们，现在我们来学习如何编写它们。

18.5.1 任务定义

在设计初期，程序员的一个主要工作实际上是选择系统中哪些活动将被定义成任务。任务的数量不应该太多。一般来说，任务越多，编程越复杂，并且对于每一次任务切换都需要时间和存储开销。

一个好的切入点是计算任务的截止时间，然后给每个任务分配一个截止时间。具有相近截止时间的活动可以设定相同的截止时间，从而将这些活动组合成一个单独的任务。功能相近并且数据交互多的活动也应该组合成一个单独的任务。

例如在图18-1的AGV寻光程序的超循环中，程序首先读取3个LDR，然后进行一些计算，之后设置电机的速度。程序还要周期性地传送数据到显示器。另外，微动开关可能在任何时刻被激活。那么这个程序需要定义多少个RTOS任务呢？一个大的原则就是集中的活动具有相近的截止时间和功能，并且处理的数据是共同的，因此集中的活动可以组成一个任务。写数据到显示器这个活动可以设置成一个单独的任务——它发生的频率比其他活动低并且优先级也相当低。由于LDR的读取将直接为电机的速度计算和速度控制提供数据，因此所有这些活动可以组成一个单独的任务。不过，LDR的读取也可以分开单独设置成一个特有的任务。最后，对微动开关的反应设置为另外一个任务。

18.5.2 编写任务以及设置任务优先级

即使在实际运行时可能会被调度器中断，作为独立或者部分独立且可以连续运行的程序也可以编写。一个任务不能调用其他任务的代码段，但是可以访问公共代码比如C语言的子例程库。一个任务可能会依赖其他任务提供的服务，也可能需要相互之间进行同步。不管属于哪种情况，RTOS都将提供一些特殊的服务来支持。

在所有情形下（但是大多情况都很简单），RTOS都允许程序员设置任务的优先级。如果任务的优先级是静态的，那么任务的优先级是固定的。如果是动态优先级，任务则会随着程序的执行而改变优先级。设置任务优先级的一种方法是衡量任务对于系统的正常运行、使用者和环境的重要性有多大。优先级可以设置为：

- ❑最高优先级：任务关系到系统存亡；
- ❑中等优先级：任务确保系统正常运行；

- 低优先级：任务完成一些不太重要的功能——这些任务可以偶尔取消或者延迟完成，并且这对于系统的运行是可以接受的。

另外还有一种方法，它是通过估算任务的截止时间来考虑任务优先级的。在这种情况下，临近截止时间的任务被设置为高优先级。然而，如果一个任务的截止时间很紧急，但是在整个任务安排中它不是很重要，那么它的优先级仍然可能会很低。

18.6 数据和资源保护——信号量

多个任务可能都需要访问相同的共享资源。共享资源可能是一个硬件（包括存储器或者外围设备）或者一个公共的软件模块。要谨慎地使用共享资源。一种处理共享资源访问的方法是使用*信号量*。信号量是分配给每个共享资源的，它用于指示资源是否被占用。

如果使用一个*二元信号量*，那么第1个需要使用该资源的任务将发现信号量处于GO状态。任务开始使用该资源之前，首先需要将信号量的状态变成WAIT状态。后面需要使用该资源的任务必须进入阻塞状态（见图18-7）。当第1个任务完成共享资源的访问，它将信号量的状态变回GO状态。这就产生了*互斥*的概念：当一个任务正在访问共享资源时，所有其他任务将被拒绝。

*计数信号量*用于一组相同的资源，比如一组打印机。现在，信号量初始时设置成资源单元的个数。当任何一个任务使用了其中一个单元，信号量将递减。当任务完成时，信号量重新递增。因此，计数信号量代表当前可用的单元数目。

由于将信号量设置成WAIT状态的结果是导致其他任务被阻塞，因此信号量可以用于提供不同任务间的时间同步和信号通知。一个任务可以通过设置信号量来阻塞其他任务；也能够设置清除信号量的时间来释放阻塞的任务。

478

在18.4.4节曾经提到过，高优先级的任务是"运动场上欺凌弱小者"。通过使用信号量，一个低优先级的任务可以在运动场上扭转局面。如果低优先级的任务设置一个资源的信号量，而高优先级的任务需要该资源，那么低优先级的任务通过设置信号量可以阻塞高优先级的任务。这将造成一个危险的情况——*优先级翻转*。这个话题超出了本书的范围，但是它在其他与实时编程相关的学习材料中有很详细的讲解，并且这是一个很值得进一步研究的话题。参考文献18.3中更详细的资料可以作为深入学习的起点。

18.7 后面我们将学习什么

现在，RTOS理论的发展已经远远超过了本章所讨论的范围，并且针对不同类型的应用领域它变得越来越专业化。由于存在很多不同的RTOS，因此只有把本章讲述的理论应用到一个真实的RTOS中，这些理论才具有意义。因此，下一章我们将会学

习一个实际的操作系统。

小结

- ❑ 几乎每一个嵌入式系统都普遍存在多任务的需求，多任务包括一些很重要的概念——任务、截止时间和优先级。
- ❑ 实时的系统操作是一个能够满足给定截止时间要求的操作。
- ❑ 通过传统的顺序编程可以实现简单的多任务实时系统。
- ❑ 更复杂的多任务实时系统需要借助于实时操作系统。
- ❑ 在实时操作系统进行应用开发有别于传统的顺序编程方式，并且它需要程序员清晰地知道操作系统的基本原理。

参考文献

18.1. Wilmshurst, T. (2002). Exploring real-time programming. *Electronics World*, pp. 54–60, January; http://www.softcopy.co.uk/electronicsworld/

18.2. A Real-Time Operating System for PICmicro™ Microcontrollers (1997). Microchip Technology Inc., 585.

18.3. Simon, D. E. (1999). *An Embedded Software Primer*. Addison-Wesley. ISBN 0-201-61569-X.

第 19 章
Salvo™实时操作系统

第 18 章只在理论层次上介绍了实时操作系统(RTOS)的概念,我们需要将这些概念应用到一个实际的操作系统中。而这个操作系统最好能运行在基于 PIC®的嵌入式系统中,因此我们选择 Salvo™实时操作系统——一个应用于小系统的操作系统。Salvo 是一个商业实时操作系统,主要是为小型嵌入式系统而设计的,而且它包含一个可运行 Microchip C18 编译器的版本。此外,还有一个免费的 Salvo LITE 版本,它可以使我们有机会进入令人兴奋但又具有挑战性的 RTOS 领域,可以方便地编写简单、直观并可以运行的程序。

本章主要介绍 Salvo 实时操作系统,让我们可以使用它编写一些实际而又简单的程序。学习本章,主要是为了了解如何将一个真正的 RTOS 应用到实际环境中,而不是变成使用 Salvo 的专家。因此,使用 Salvo 的具体细节可以参考用户指南。

本章主要围绕 3 个例程来阐述一些 RTOS 关键概念的应用。在本章你将学到:

☐ Salvo 实时操作系统的一些基础知识;

☐ 从实用角度考察一个实际 RTOS 的工作方式;

☐ 使用 RTOS 的优点和缺点。

19.1 Salvo 实时操作系统概述

最初,使用汇编语言开发的 Salvo 是用于赛车的数据采集系统。当得到广泛的应用之后,人们使用 C 语言对它进行重写,并使它适用于一般性应用。它的目标应用领域是小型的嵌入式系统。现在,Salvo 需要 C 编译器运行,而不再需要汇编器。Salvo 有多个版本适用于许多主流的嵌入式系统编译器。它是由 Pumpkin Real Time Software 公司提供的。

为了满足编写一些有趣的入门级程序的需要,后续部分将有选择性地介绍 Salvo 的一些知识。关于 Salvo 的全部细节,你可以参考供应商提供的参考文档——主要是参考文献 19.1~19.3。

19.1.1 Salvo 的基本特性

Salvo 可以运行多个不同优先级的任务,并且采用协作式调度策略进行任务调度。由于协作式调度对存储空间要求不高,从而可以满足低存储空间的要求,在 18.4.5 节

已经讨论过。任务的数目(在全功能版本中)只受限于 RAM 的大小,但是任务只能设置 16 个优先级。不同的任务可以具有相同的优先级。Salvo 支持许多不同的"事件",包括二元信号量和计数信号量、消息和消息队列。

Salvo 由很多文件组成——源文件、头文件、库文件和其他文件。实际上,这些文件以提供额外服务的形式供主编译器使用。通常情况下,程序员可以直接使用主编译器来满足编译需求,也可以在需要时给编译器指定 Salvo 文件。在程序开发过程中,程序员当然必须遵循 Salvo 的要求。最后,由程序员开发出来的程序包含了程序源代码、Salvo 和编译器的头文件和源文件、Salvo 和编译器的库文件。图 19-1 总结了构建过程和主要用到的文件。连接器输出的是一个可以下载到程序存储器的可执行文件。

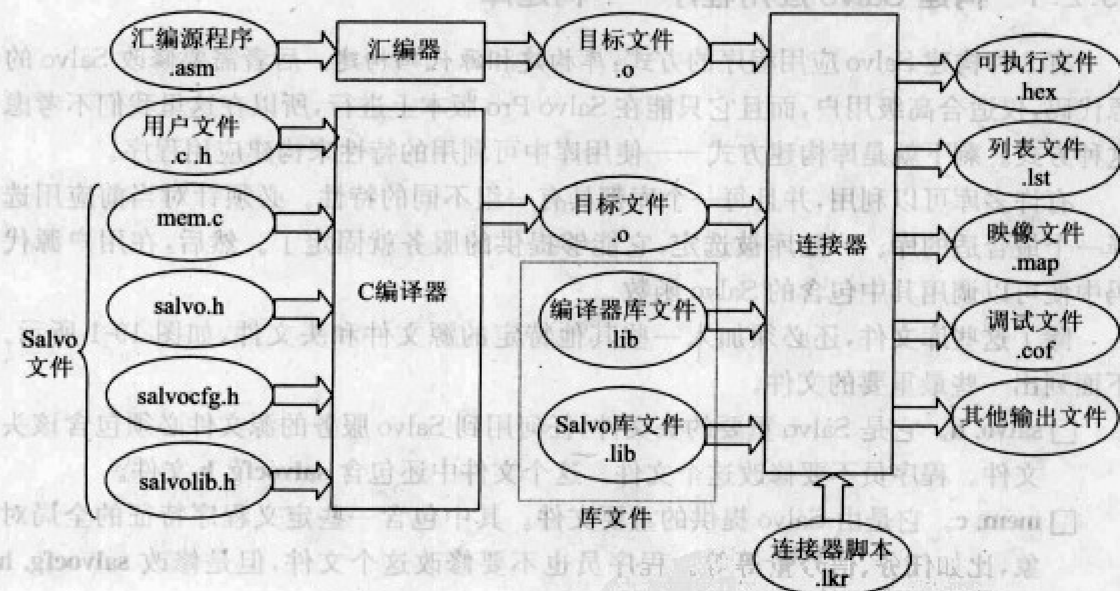


图 19-1 Salvo 构建过程

19.1.2 Salvo 版本和相关的参考文献

Salvo 有许多可用的版本。豪华版 Salvo Pro 的特点是高配置并且具有全部特性。免费版 Salvo Lite 包含 Salvo Pro 版的部分功能。它可以从 Pumpkin 公司的网站:http://www.pumpkininc.com/下载,也可以从本书附属资源中获得。选择 Salvo 版本时,必须与当前使用的编译器匹配——本书使用的是 Microchip C18 编译器。免费版 Salvo Lite 只允许最多 3 个任务和 5 个事件。从这一点看,相对于全功能版本而言,该免费版本存在一定的限制。而实际上,使用它可以开发出非常有用而高级的程序。

481

Salvo 提供了大量易读的用户手册[19.1]。手册支持所有版本的 Salvo,从免费版到 Salvo Pro。对于只是使用 Salvo Lite 版本的初学者来说,手册中的有些内容可能比较难。不过,其中有一些很好的而且内容丰富的介绍性章节。手册中包括所有 RTOS 的一般资料,其中的内容与编译器无关。另外一些参考手册介绍了与编译器相关的

RTOS。对于 Microchip C 编译器，比较重要的是(截至本书编写时)参考文献 19.2 和参考文献 19.3。

19.2 配置 Salvo 应用程序

Salvo 的一个基本特征是高度可配置性。因此，有必要尽早阐述一下这些配置。本节将介绍 Salvo 文件和配置的基本特点，这将有助于我们更好地理解图 19-1 中的构建过程。

19.2.1 构建 Salvo 应用程序——构建库

有 2 种构建 Salvo 应用程序的方式：*库构建和源代码构建*。后者需要修改 Salvo 的源代码，仅适合高级用户，而且它只能在 Salvo Pro 版本上进行，所以在这里我们不考虑这种方式。剩下就是库构建方式——使用库中可利用的特性来构建应用程序。

有许多库可以利用，并且每一个库都具有一组不同的特性。必须针对当前应用选择一个最合适的库。一旦库被选定，它能够提供的服务就固定了。然后，在用户源代码中便可以调用其中包含的 Salvo 函数。

除了这些库文件，还必须加入一些其他特定的源文件和头文件，如图 19-1 所示。下面列出一些最重要的文件。

- **salvo.h**。它是 Salvo 重要的头文件，任何用到 Salvo 服务的源文件必须包含该头文件。程序员不要修改这个文件。这个文件中还包含 **salvocfg.h** 文件。
- **mem.c**。它是由 Salvo 提供的重要文件。其中包含一些定义程序特征的全局对象，比如任务、信号量等等。程序员也不要修改这个文件，但是修改 **salvocfg.h** 文件的内容会间接影响它。
- **salvocfg.h**。这个文件由程序员编写，其中包括程序员根据应用对 Salvo 所做的许多配置。它设置了系统某些关键特性，比如使用哪一个库、有多少个任务和事件。详细内容在 19.4.4 节讲述。

19.2.2 Salvo 库

Salvo 包括大量的标准库，其中包含 RTOS 的许多功能。针对不同的编译器以及编译器的不同版本，对库有不同的配置。Salvo 支持的不同库配置包括不同的存储器模型和不同特性。配置 Salvo 应用程序的一个技巧是对库的选择——库中最好只支持应用所需要的特性，而不支持任何其他特性。

Salvo Lite 提供了一些免费库。它们像标准库，但是在功能上有限制。Salvo 库(针对 C18 编译器)的命名方法如图 19-2 所示。库名的最后一个字母指定库的“配置”，具体含义在表 19-1 中说明。比如，**sfc18sfa.lib** 库支持除了超时以外的所有特性。特性多会导致库比较大。如果在应用中只需要多任务特性，最好使用“m”库。这将获得更高

的编码效率，且所需的存储空间更少。



图 19-2 Salvo 库命名方法，针对 C18 编译器

表 19-1 不同库配置下可用的库服务

	库配置				
	a	d	e	m	t
多任务	+	+	+	+	+
延迟	+	+	−	−	+
事件	+	−	+	−	+
空转	+	+	+	−	+
任务优先级	+	+	+	−	+
超时	−	−	−	−	+

＋＝启用；－＝失效

19.2.3 C18 和 Salvo 版本

当使用 Salvo 进行应用开发时，很有必要确保 MPLAB® IDE、C18 编译器和 Salvo 三者的协调一致。本书使用的是 MPLAB 7.22、C18 3.00 和 Salvo3.2.3.c。应该从 Salvo 网站下载 Salvo 安装文件，并按照通常的过程来安装。安装过程很简单，参考文献 19.1 给出了详细说明。安装会产生一些文件夹，如图 19-3 所示。

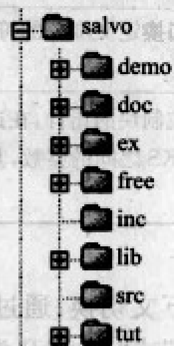


图 19-3 Salvo Lite 文件夹

19.3 编写 Salvo 程序

483

本节会首先介绍如何进行 Salvo 编程，之后再列举一个例程。

19.3.1 初始化和调度

Salvo RTOS 的许多特性体现在 C 语言库函数上。可以通过熟悉这些函数以及它们的功能培养 Salvo 编程的技巧。表 19-2 给出了一些示例函数，这些函数正好也会在第一个例程中使用到。表中给出了函数或服务的名称并且总结了它们的操作和所携带的参数(如果有的话)。在后面几页，将重复地引用和解释表中内容——如果暂时不清楚这些函数的详细内容，也没有关系。

表 19-2 核心 Salvo 服务示例

函数/服务	操作和参数
OSInit()	初始化操作系统，包括数据结构、指针和计数器。在使用任何其他 Salvo 函数之前，必须调用它 无参数
OSSched()	Salvo 调度器。每次调用，该函数将从所有就绪任务中选择一个任务运行。反复调用会产生多任务 无参数
OS_Yield()*	程序无条件返回调度器 上下文切换符号，通常使用_OSLabel()函数来定义
OSCreateTask(,,)	创建一个任务并且启动它(即设置任务为就绪状态) (1)任务的起始地址指针——通常为任务名称 (2)任务的 TCB(任务控制块，Task Control Block)指针 (3)优先级——编号 0(最高)~15(最低)
OSStartTask()	将一个停止的任务变为就绪 任务的 TCB(任务控制块，Task Control Block)指针
_OSLabel()	为每个上下文切换定义唯一的符号 符号名称
OSTCBP()	定义指向特定控制块的指针，在这里即指任务控制块 从 1 到 OSTASKS 之间的整数，其中 OSTASKS 指定了任务的个数，在 salvocfg.h 文件中定义

484

* 引发上下文切换。

表中的一些函数可以引发上下文切换；通过调用它们可以实现协作式调度。所有用户可调用的 Salvo 函数均以“OS”或“_OS”开头。以“_OS”开头的函数则包含一个有条件或无条件的上下文切换。

为了启动和初始化 RTOS，必须在调用任何其他 Salvo 函数之前调用 OSInit()函

数。然后调用 OSCreateTask()函数来创建任务,而且要确保正确设置该函数的参数。

OSSched()函数作为 Salvo 的任务调度器。每一次调用,它会做以下 3 件事。

- ❑选择一个最合格(即优先权最高)的任务,并运行它。当有多个任务具有相同的优先级时,使用轮转法来处理。
- ❑如果有事件发生,就处理事件队列。回忆一下,事件包括信号量和消息。这个操作可能使某些任务变为就绪状态。
- ❑处理延迟队列。在 Salvo 中,延迟是一个控制任务运行的重要方法。这个操作也可能使某些任务变为就绪状态。

19.3.2 编写 Salvo 任务

一个 Salvo 任务就是一个 C 语言函数,而且要遵循 18.5 节中所述的任务编写的一般原则。具体针对 Salvo 来说,还有一些重要的要求。这些要求如下所述。

- ❑所有的任务最初是被"销毁的"。必须调用 OSCreateTask()函数来创建任务。任务可以在程序的任何地方被创建。在实际应用中,许多任务是早在 main 函数中就已创建。
- ❑通常,在任务中可能有一段初始化,随后就是一个无限循环。在循环中必须至少包含一个上下文切换。
- ❑调用 OS_Yield()函数可以提供上下文切换,尽管也有一些其他的函数能引发上下文切换。通过该函数(或者其他类似函数)调用,任务放弃 CPU 并移交控制权给调度器。这是使用 Salvo 进行协作式调度的基础。
- ❑任务不能携带任何参数。
- ❑在任务中,使用 static 类型的变量。这样当任务停止运行时,数据不会改变。如果数据在上下文切换之后不需要保持原值,那么可以使用 auto 类型的变量。

任务的操作特性包含在任务控制块(TCB)中。它是唯一分配给每个任务的一个存储块,其中包括任务的起始地址、状态和优先级等。

一般地,任务的状态变化遵循图 18-7 中的状态变迁图。后面将会针对 Salvo 介绍每个状态的含义。

19.4 第一个 Salvo 例程

第一个基于 Salvo 的程序如例程 19-1 所示。它使用了表 19-1 中给出的 Salvo 服务。程序中只包含 2 个相同优先级的任务。**Count_Task** 任务的功能是递增一个计数器的值。另外一个任务 **Disply_Task** 的功能是显示计数器数值的两位,这两位连接到端口 C 的 2 个比特位。由于程序可以在 AGV 硬件上运行,计数器最低 2 个有效位左移之后用来控制 AGV 上的 LED(LED 连到端口 C 的位 5 和位 6)。

例程 19-1 第一个 Salvo RTOS 应用程序

```
/*************************************************************************
rtos_ex1                                     An introductory Salvo example.

There are two tasks,of equal priority.One counts,the other displays the count.
Salvo Lite RTOS with sfc18sfm.lib library used.
Mainly for simulation but can run on Derbot.
TJW 21.12.05                                              Tested 21.12.05
*************************************************************************/

#include <salvo.h>
#undef OSC    //necessary for this Salvo version, as it also defines this name
#include  <p18f242.h>

#pragma config OSC = HS, OSCS = OFF  //HS oscillator, oscillator switch off
#pragma config PWRT = ON, BOR = OFF  //pwr-up timer on, brown-out detect off
#pragma config WDT = OFF             //watchdog timer off
#pragma config STVR = ON, LVP = OFF  //Stack overflow reset enable on,
                                        //low voltage programming off

//function prototypes. These functions are tasks.
void Count_Task( void );
void Display_Task( void );

//Define labels for context switches
_OSLabel(Count_Task1)
_OSLabel(Display_Task1)

//Define and initialise variable
unsigned char counter = 0;

/*************************************************************************
Task Definitions (configured as functions)
*************************************************************************/
void Count_Task( void )
{
    for (;;)                        //infinite loop
    {
    counter++;
    OS_Yield(Count_Task1); //context switch
    }
}
//
void Display_Task( void )
{
    for (;;)
    {
    PORTC = counter<<5; //Shift Counter left, and move to PORT C
OS_Yield(Display_Task1);
}
```

```
}
/************************************************************************
Main
************************************************************************/
void main( void )
{
//Initialise
      TRISC = 0b10000000; //Setall Port C bits to output, except bit 7.
      PORTC = 0;          //Setall Port C outputs low
//Initialise the RTOS
    OSInit();
//Create Tasks
    OSCreateTask(Count_Task, OSTCBP(1), 10);
    OSCreateTask(Display_Task, OSTCBP(2), 10);
 //Set up continuous loop, within which scheduling will take place.
    for (;;)
    OSSched();
}
```

19.4.1 程序的总体结构和 main 函数

初次浏览这个例程的代码清单，它看似是一个常规的小型 C 程序。首先发现它是 Salvo 程序的位置是在源代码中 **salvo.h** 文件的包含处。接下来几行，**_OSLabel** 宏被应用了 2 次。这个宏用于定义上下文切换符号。程序只包含 2 个上下文切换，每个任务包含一个。我们后面将会看到这两个符号 **CountTask1** 和 **DisplayTask1** 是如何应用于任务的上下文切换的。

main 函数以传统方式开始，它只是对程序中唯一用到的端口 C 做了一些初始化的工作。然后，程序调用 **OSInit( )** 函数，它的作用是初始化操作系统和设置后续所有 RTOS 操作需要的操作环境。随后，程序通过 2 次调用 OSCreateTask() 函数来创建了 2 个任务。表 19-1 总结了这个函数调用的格式——必须提供 3 个入口参数。任务的开始地址就是任务名称，而任务名称实际上就是任务的函数原型。TCB 的开始地址由 OSTCBP() 宏来定义。宏的参数从 1 开始向上递增，它为每个任务分配一个 TCB 块。随后，2 个任务被设置成相同的优先级。可以选择 1～16 之间的任何数作为优先级，这里随意选择一个优先级 10。函数最后是一个无限循环，在循环中程序重复地调用 OSSched() 调度器。这个函数将会激活最合格的任务。

在程序中，配置位的设置是采用第 17 章中所描述的方式。但是，这种方式会导致与当前 Salvo 版本发生冲突，因为在 salvo.h 和 p18F242.h 这 2 个文件中都定义了名称 OSC。Pumpkin 公司是这样建议的：对于 C18 编译器来说，操作系统可以不需要定义 OSC。因此，在 p18F242.h 文件包含语句的前一行，使用 # undef 预处理机伪指令取消 Salvo 的 OSC 定义。

19.4.2 任务和调度

我们可以看到任务本身是被编写成函数形式的。每个任务都是一个包含 **OS_Yield()** 调用的无限循环。**OS_Yield()** 的入口参数是上下文切换符号，这在前面已经定义。每次任务被激活时，它将一直运行直到程序执行到 **OS_Yield()** 处。在这里，任务将控制权返还给调度器，并且它的状态由"运行"变成"就绪"或"合格"（如图 18-7 所示）。当任务再次被激活时，它将从 **OS_Yield()** 之后的一行语句继续执行并且返回到"运行"状态。

19.4.3 创建一个 Salvo/C18 项目

Salvo 项目的创建最初与任何 C18 项目都是一样的。在这里简要介绍的创建步骤类似于参考文献 19.3 中的描述，但是文献中还针对出现的一些问题给出了相应的解决建议。这是一个针对 MPLAB 6 的应用文档，但是它同样可适用于 MPLAB 7.22 版本。这是由于截至本书编写时，还没有针对 MPLAB 7.22 版本的应用文档可以利用。

如果你想要自己构建这个项目（强烈建议你亲自实践），可以首先按照正常方式创建一个 MPLAB 项目。在这里，我们给项目起名为 **rtos_ex1**。并且为这个项目新建一个文件夹（它将包括许多文件），然后从本书附属资源中将例程 19-1 中的源文件和 **salvo.h** 复制到刚才新建的文件夹中。之后，从 C18 目录中，将 18F242 连接器脚本添加进来，并从 Salvo 目录中，将 **sfc18sfm.lib** 库文件也添加进来。再从 Salvo 目录中，添加 **mem.c** 文件。最后，你的 MPLAB 项目窗口应该如图 19-4 所示。

回顾图 19-2 和表 19-1，我们来看一下 **sfc18sfm.lib** 库的特性。不难发现它是一个针对 C18 编译器的免费库，它支持小型的存储器模型并且只具有多任务的功能。对于这个非常简单的程序来说，选择使用这个库是合适的。

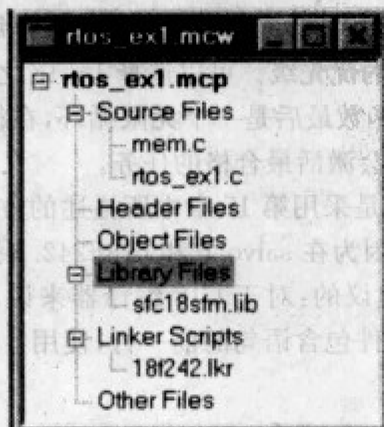


图 19-4 Salvo_Ex1 项目中使用的文件

19.4.4 配置文件的设置

通过修改 **salvocfg.h** 文件,程序员可以配置 Salvo RTOS,从而使 Salvo 适合某个特定的应用。**salvocfg.h** 文件由一系列的 C 语言定义语句组成。Salvo Lite 只能利用其中一个受限的功能子集,而全功能版的 Salvo 可以支持更多特性。每个配置选项设置有一个默认值,针对实际应用可以修改其中某些(或者全部)配置选项的值。但是在修改时必须注意:配置文件中的设置越接近实际应用,最终生成的代码才会越高效。比如,当设计中只有 3 个任务时,存储空间就不应该配置成 6 个任务要求的那么大。

项目 **rtos_ex1** 使用了 **salvocfg.h** 配置文件,如例程 19-2 所示。程序中每一行配置都有注释用于解释它的含义。可以发现,这个简单的配置文件中的配置选项只与库配置和程序特性有关系,而后者包括任务、事件和消息。第 1 行是一个基本配置,它用来选择当前要使用的预编译库。库的配置必须匹配当前使用的库,这一点非常重要。由于当前使用的是 **sfc18sfm.lib** 库,因此使用了 OSM 库配置。

例程 19-2 rtos_ex1 项目中使用的 salvocfg.h

```
/*************************************************************************
salvocfg.h file for rtos_ex1

TJW 8.1.06
*************************************************************************/
//Library configuration
#define OSUSE_LIBRARY          TRUE   //Use precompiled Salvo library
#define OSLIBRARY_TYPE         OSF    //use freeware library
                                          //(OSL is standard library)
#define OSLIBRARY_GLOBALS      OSF    //Salvo objects far, in banked RAM
#define OSLIBRARY_CONFIG       OSM    //Set library configuration,
                                         //OSM = support multitasking only
#define OSLIBRARY_VARIANT OSNONE.//No library variant

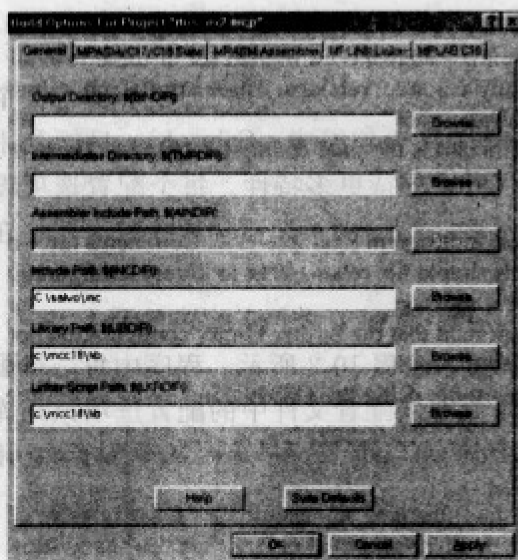
//Tasks, Events and Messages Configuration
#define OSEVENTS               0      //define maximum number of events
#define OSEVENT_FLAGS          0      //define maximum number of event flags
#define OSMESSAGE_QUEUES       0      //define maximum number of message queues
#define OSTASKS                2      //define maximum number of tasks
```

19.4.5 构建 Salvo 例子

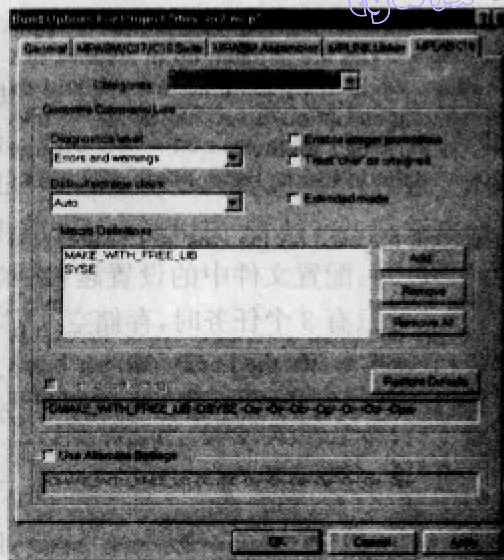
构建 Salvo 项目的过程与构建 C18 项目是相同的。但是为了正确构建 Salvo 项目,还有更多的事情要做。除了解决所有可能的编程和连接错误之外,还必须正确配置 Savlo 项目。

图 19-4 显示了项目中所有需要使用到的文件。必须在 Build Options 对话框中正确设置 Salvo 包含文件的搜索路径,如图 19-5a 所示。并且还必须设置图 19-5b 中的配置选项。**SYSE** 选项用于 Salvo 识别 C18 编译器,其中包含了一些为多个编译器编写的通用文件。

489



(a) General



(b) MPLAB C18

图 19-5 设置 Salvo 项目的构建选项

如果使用的是本书附属资源中提供的文件，那么构建过程应该不会出现问题。但是如果仍然出现了问题，你可以查阅参考文献或者登录 Salvo 用户论坛（在 Pumpkin 公司的网站上），也可以查看本书网站（www.embedded-knowhow.co.uk）。

19.4.6 仿真 Salvo 程序

第 1 个 RTOS 例程的仿真方法与仿真其他任何程序是一样的。当成功构建出程序之后，选择 MPLAB SIM 仿真器。

打开一个 Watch 窗口，并且选择显示 **counter** 和 **PortC** 变量。插入图 19-6 所示的断点。这些断点均设置在这个简单程序的关键地方。然后运行仿真器到第 1 个断点，该断点正好在 RTOS 初始化操作之前。继续运行程序，它将停在 OSSched()处。再次运行，程序开始执行任务 Count_Task。由于 Count_Task 任务首先被创建，因此先运行它。如果继续从这里开始单步执行程序，你会发现在任务中程序是按顺序执行的，直到到达 **OS_Yield()** 调用处。

继续按下运行命令将会引发程序在不同的任务间交替执行，而且不同任务的执行之间会调用一次 OSSched()函数。每次轮到任务开始执行，它都正好接着离开时的位置继续执行。由于任务的优先级相同，因此这些任务以轮转调度的方式交替执行。

490

每次运行 Count_Task 任务，Watch 窗口的 **counter** 值都会增加；每次运行 Display_Task 任务，端口 C 的值都会被更新。从程序功能来看，我们实现的功能并不新奇。但从程序的执行方式来看，它却是全新的。

```
void Count_Task( void )
{
   for (;;) {                    //infinite loop
     counter++;
     OS_Yield(Count_Task1); //context switch
   }
}
//
void Display_Task( void )
{
   for (;;) {
  PORTC = counter<<5;              //Shift Counter left, and move to PORT C

         OS_Yield(Display_Task1);
   }
}
/**************************************************************
Main
***************************************************************/
void main( void )
{
//Initialise
    TRISC = 0b10000000; //Set all Port C bits to output, except bit 7.
    PORTC = 0;           //Set all Port C outputs low
//Initialise the RTOS
    OSInit();
//Create Tasks
    OSCreateTask(Count_Task, OSTCBP(1), 10);
    OSCreateTask(Display_Task, OSTCBP(2), 10);
 //Set up continuous loop, within which scheduling will take place.
    for (;;)
       OSSched();
}
```

图 19-6 RTOS 仿真时的断点设置

你也可以在实际的 Derbot 硬件上运行这个程序。然而，程序的运行结果并不是很有趣。这是由于在程序中没有控制任务的执行速度，因此 LED 灯一直处于点亮状态。

19.5 在 Salvo 程序中使用中断、延迟和信号量

任何实时操作系统的一个关键特征当然是它管理实时活动的能力。因此，以固定可靠的频率在系统中产生一个连续的“时钟滴答”几乎是必需的。时钟滴答是一个时基，它是其他事件何时发生的依据。一旦产生了时钟滴答，许多 Salvo 的特性就可能发挥出来。

我们利用定时器中断来产生时钟滴答。然后我们需要一些用于计数和对延迟做出反应的服务，而这些服务都是在这个时钟滴答的基础上工作的。表 19-3 给出了同中断和定时相关的一些 Salvo 服务，后面我们将会用到它们。

表 19-3 使用中断、定时器和延迟的 Salvo 函数和服务示例

函数/服务	操作和参数
OSTimer()	检查所有处于延迟或者等待的任务是否已超时。如果是，设置这些任务为就绪状态。如果系统需要延迟、超时或者与计时相关的服务，必须以时钟滴答的频率调用该函数。通常，该函数被放在定时器的中断服务程序中 无参数
OS_Delay(,)*	停止当前任务运行，返回控制权到调度器并且任务延迟指定的一段时间。需要同 OSTimer()函数配合使用 (1)以系统的时钟滴答为单位的整数，指定任务的延迟时间(8 位数值) (2)上下文切换符号，通常使用_OSLabel()宏来定义
OSEi()	启用中断(设置 INTCON 寄存器中的 GEI 和 PEIE 位，见图 12-8)
OSDi()	禁止中断

*引发上下文切换。

19.5.1 一个使用中断驱动的时钟滴答的例程

例程 19-3 命名为 **rtos_ex2**，它是对第一个 RTOS 例程的扩展。它仍然只包含 2 个相同优先级的任务，但引入了 1 个中断驱动的时钟滴答、1 个延迟和 1 个二元信号量。随后，我们将详细介绍它们。你可以在例程源代码中找到用来产生时钟滴答的中断服务程序。

虽然增加了一些新成分，但是程序结构还是很清晰的。**main** 函数通过调用 **Micro_Init()** 来初始化微控制器。这种做法是仿效 RTOS 调用 **OSInit()** 进行初始化。在创建了任务和信号量之后，程序进入调度循环。

例程 19-3 应用延时和信号量

```
/*****************************************************************************
rtos_ex2
                                              A further Salvo example.

Applies Timer interrupt, clock tick, delays, and binary semaphore.
There are two tasks,of equal priority.One counts,the other displays the count.
Salvo Lite RTOS, with Library sfc18sfa used.
Can be simulated, or run on Derbot.
TJW 28.12.05                                                 Tested 30.12.05
*****************************************************************************/

#include <salvo.h>
#undef OSC //necessary for this Salvo version, as it also defines this name
#include <p18f242.h>
#include <timers.h>
```

492

```
#pragma config OSC = HS, OSCS = OFF  //HS oscillator, oscillator switch off
#pragma config PWRT = ON, BOR = OFF  //pwr-up timer on, brown-out detect off
#pragma config WDT = OFF             //watchdog timer off
#pragma config STVR = ON, LVP = OFF  //Stack overflow reset enable on,
                                        //low voltage programming off

#define BINSEM_Display OSECBP(1)

//function prototypes.
void Micro_Init(void);

//These functions are tasks.
void Count_Task( void );
void Display_Task( void );

//Define and initialise variable
unsigned char counter;

//Define labels for context switches
_OSLabel(Count_Task1)
_OSLabel(Display_Task1)

/*****************************************************************************
User-defined Functions, including RTOS Tasks.
*****************************************************************************/
void Count_Task(void)
{
    for (;;) {                       //infinite loop
      counter++;
     OSSignalBinSem(BINSEM_Display);
     OS_Delay (20,Count_Task1);      //Task switch, delay for 20x10ms, (200ms)
                                     //Use smaller delay for simulation
    }
}

void Display_Task(void)
{
    for (;;) {                      //infinite loop
    OS_WaitBinSem(BINSEM_Display, 100, Display_Task1);
    PORTC = counter<<5;             //Shift Counter left, and move to PORT C
    OS_Yield(Display_Task1);
    }
}
void Micro_Init(void)
{
//Initialise Port C
     TRISC = 0b10000000;
     PORTC = 0;                     //Switch outputs off
/*Initialise TMR0: interrupt enabled,16-bit operation, internal clock,
prescaler divide by 16, hence (with 4MHz clock) input cycle period of 16us*/
   OpenTimer0 (TIMER_INT_ON & T0_SOURCE_INT & T0_16BIT & T0_PS_1_16);
  counter = 0;
}
```

```
/******************************************************************************
Main
******************************************************************************/
void main( void )
{
//Initialise Microcontroller
       Micro_Init();
//Initialise RTOS
    OSInit();
    OSCreateTask(Count_Task, OSTCBP(1), 10);  //Create the Count_Task Task
    OSCreateTask(Display_Task, OSTCBP(2), 10); //Create the Display_Task Task
    OSCreateBinSem(BINSEM_Display, 0);         //Create the Binary Semaphore
//Enable interrupts
    OSEi();
//Scheduling Loop
    for (;;)
        OSSched();
}
```

19.5.2 选择库和配置

前面的例程 **rtos_ex1** 中使用了 **sfc18sfa.lib** 库。但是，现在我们引入了延时和信号量，而后者是一个"事件"类型。因此，从表 19-1 中我们发现后缀为 m 的库已经不适合了。我们能够选择的最适合的库只能是后缀为 a 的库，尽管它也具备我们不需要的空转和优先级的特性。

现在项目使用了一个不同的库并且库包括新的特性，因此我们需要一个新的 **salvocfg.h** 文件。除了有 2 个地方——修改的库配置、包含一个信号量（属于"事件"）——不同之外，这个文件基本上与例程 **rtos_ex1** 中的是相同的。因此，下面几行可以插入到前面 **salvocfg.h** 文件中相同位置的行。

```
#define OSLIBRARY_CONFIG OSA  //Set library configuration,
                              //OSA = support multitasking, delays & events
...
#define OSEVENTS          1   //define maximum number of events
```

19.5.3 使用中断和产生时钟滴答

通过在宏定义窗口（图 19-5b）中增加 **USE_INTERRUPTS** 定义，可以在 Salvo 程序中引入中断。这个定义与作为系统中断服务程序的 **isr.c** 文件一起实现了 Salvo 的中断功能。中断服务程序是以正常方式来编写的，如 17.4 节所述。

例程 19-4 是项目 **rtos_ex2** 的中断服务程序。它遵循例程 17-3 的结构，但是其中调用了一个非常重要的函数 **OSTimer()**。从表 19-3 中可以发现，这个函数用于配合其他 Salvo 函数发挥 Salvo 的时基特性。每次调用 ISR，变量 **tick_counter** 会递增。这个变量主要用于程序的仿真。

这个例程中设定（有一些随意性）时钟滴答的周期为 10ms。它是利用 Timer 0 的

溢出中断来产生的。通过调用 **OpenTimer0()** 函数来启用和配置 Timer 0，你可以在源代码的注释中看到对 Timer 0 设置的描述。由于时钟振荡器频率为 4MHz 并且预分频比设置为÷16，所以输入到定时器的时钟周期为 16μs。因此，理论上定时器需要 625 个时钟周期才能产生一个 10ms 的中断周期。于是定时器的重载入值需要设置为 (65536−625)，即 64911。当在仿真程序时通过 MPLAB SIM 的 Stopwatch 功能对中断延迟进行测量之后，考虑到延迟时间是不可忽略的，我们会增加定时器的重载入值。

程序是调用 **OpenTimer 0()** 函数来启用定时器中断的。而全局中断使能是通过在 **main** 函数中调用 **OSEI()** 来设置的。**OSEI()** 并不是 Salvo 相关的函数，在这里使用它只是为了方便。

例程 19-4 产生"时钟滴答"的中断服务程序

```
/**************************************************************************
ISR for rtos_ex2
Timer 0 interrupt is high priority source.
Reloads Timer, and calls OSTimer()
TJW 30.12.05                                              Tested 30.12.05
**************************************************************************/

#include <salvo.h>
#include <p18F242.h>
#include <timers.h>

//function prototype(s)
void timer0_isr (void);

static unsigned int tick_counter = 0;

//Define the high priority interrupt vector to be at 0008h
#pragma code high_vector=0x08

void interrupt (void)
{
  _asm GOTO timer0_isr _endasm //jump to ISR
}
#pragma code //Return to default code section

//Function timer0_isr specified as high-priority ISR
#pragma interrupt timer0_isr

//timer0_isr function.
void timer0_isr (void)
{
      WriteTimer0 (64918); //Reload value gives 625 cycles to overflow,
                           //less compensation for interrupt latency
      OSTimer();
      tick_counter++;      //increment tick counter, (for simulation)
      INTCONbits.TMR0IF = 0; //Clear TMR0 interrupt flag
}
```

尽管现在我们产生了一个有用(也可以认为是必要的)的时钟滴答,但是我们需要注意通过这种方式产生的时钟滴答,其周期是近似精确的。这是因为 RTOS 会经常禁止中断,比如在调用 **OSSched()**期间。这将加大中断延迟,从而延缓 CPU 对定时器的中断服务程序做出反应的速度。如果定时器设置为硬件自动重载,它将会提供一个更精确的时基。但是不管定时器如何设置,由于 Salvo 使用的是协作式调度策略,这样会导致只有在其他任务允许的条件下,任务才能对时钟滴答做出反应,因此形成的时基还是近似精确的。后面在程序仿真时,我们会能够估计出这种近似的程度。

19.5.4 使用延迟

既然已经产生了一个时钟滴答,我们就可以利用它来进行任务同步。一种方法是使用 **OS_Delay()**函数,它的入口参数如表 19-3 所示。该函数的调用将强制进行一个上下文切换,因此它可以替代 **OS_Yield()**函数。它还能设定任务再次运行的时间延迟,这是由函数的第一个入口参数确定的。

OS_Delay()函数的调用方法可以在例程 19-3 中的 **Count_Task** 函数中看到,如下所示:

```
OS_Delay (20,Count_Task1); //Task switch, delay for 20x10ms, (200ms)
                           //Use smaller delay for simulation
```

正如注释所描述的,这个延时使得 **Count_Task** 函数每 200ms 执行一次。

19.5.5 使用一个二元信号量

信号量的概念已经在 18.6 节介绍过了。它可以用于资源保护或者任务间的信号通信。信号量的作用很强大,它实际上是一种最简单的任务间建立通信的方式。例程 19-3 使用了一个二元信号量。

如同任务一样,Salvo 信号量必须首先在程序中创建。创建之后,每个事件将会在存储器中分配一个事件控制块(Event Control Block,ECB),类似于任务的 TCB。信号量的一些基本信息存储在其中。然后,一个任务设置信号量而另外一个任务用来等待它。完成创建、设置和等待信号量这 3 个功能的函数在表 19-4 中列出。

例程中只使用了一个信号量。由于它是程序中唯一的信号量,所以分配给它的 ECB 指针为 **OSECB(1)**。由于在函数调用中,ECB 指针使用的次数比 TCB 指针多,所以给它定义一个名称是必要的。因此,使用宏定义将它命名为 **BINSEM_Display**,如下面程序行:

```
#define BINSEM_Display OSECBP(1)
```

在 **main()**函数中,当任务被创建之后,这个信号量立即就被创建了,在如下程序行中:


```
OSCreateBinSem(BINSEM_Display, 0);
```

这个函数第一个参数是前面已定义的 ECB 指针。信号量初始值设置为0。

表 19-4 用于二元信号量的 Salvo 函数和服务示例

函数/服务	操作和参数
OSCreateBinSem(,)	创建一个二元信号量 (1) ECB 指针 (2) 初始值(0 或 1)
OSSignalBinSem()	通知二元信号量。如果没有任务等待这个信号量,其值增加。如果有任务在等待,那么具有最高优先级的任务将变为就绪状态 信号量的 ECB 指针
OS_WaitBinSem(,,)*	任务等待(处于"等待"状态)直到一个二元信号量被通知。当任务是最高优先级并且信号量被通知或者任务的暂停时间结束,那么任务将退出等待状态。然后信号量被自动清除。调用该函数必须指定任务的暂停时间 (1) ECB 指针 (2) 暂停时间(以系统的时钟滴答为单位) (3) 上下文切换符号
OSECB()	定义指定控制块的指针,在当前情况下就是指事件控制块 类似于表 19-2 中的 OSTCB() 从 1 到 OSEVENTS 之间的整数,其中 OSEVENTS 指定了事件的个数,它被定义在 salvocfg. h 文件中
OSNO_TIMEOUT	如果任务暂停时间已经结束,就用这个预定义的暂停时间

* 引发上下文切换。

通过分析程序中 2 个任务的活动,可以间接地理解这个二元信号量实际的工作机制。首先要注意,由于使用了 **OS_Delay()** 函数,每 20 个时钟滴答 **Count_Task** 任务执行一次。当 **Count _Task** 任务开始执行,它将调用 **OSSignalBinSem()** 函数来通知信号量。这个动作将设置信号量为高。如果有任务等待这个信号量,那么等待的任务将变为就绪状态以准备运行并且信号量被清除。

同时,**Display_Task** 任务正在等待该信号量变高,在如下程序行中:

```
OS_WaitBinSem(BINSEM_Display, 100, Display_Task1);
```

可以看到这个函数携带的参数:等待的信号量、任务的暂停时间和上下文切换符号。任务的暂停时间必须指定,即使任务不会执行这个暂停(在本例中)。整个程序运行的效果是只要计数值在 **Count _Task** 任务中被更新,那么紧接着就会执行任务 **Display_Task** 来显示这个计数值。

通过上面的几个步骤,我们实现了一个简单方式的任务间通信和同步。这在编程上是一个巨大的进步,而且它还有一个优点:正在等待信号量的任务是不能被调度器激活的,因此,CPU 可以获得更有效的利用。

19.5.6 程序仿真

仿真这个程序是很有趣的，并且可以实际地看一下时钟滴答、任务和信号量三者是如何交互的。从本书附属资源中复制出源文件和 **salvocfg.h** 文件，然后创建和构建项目。

497

在仿真时，建议使用下面推荐的仿真设置。选择 MPLAB SIM 作为调试器并且设置 3 个断点：在 2 个任务中都设置 1 个断点、在 ISR 处设置 1 个断点。打开 Watch 窗口，并加入图 19-7 中所示的变量。从工具条中，选择 **Debugger > Settings> Osc/Trace** 并且设置晶振的频率为 4MHz。从 Debugger 的下拉菜单中打开 Stopwatch 窗口，如图 19-7上部分所示。

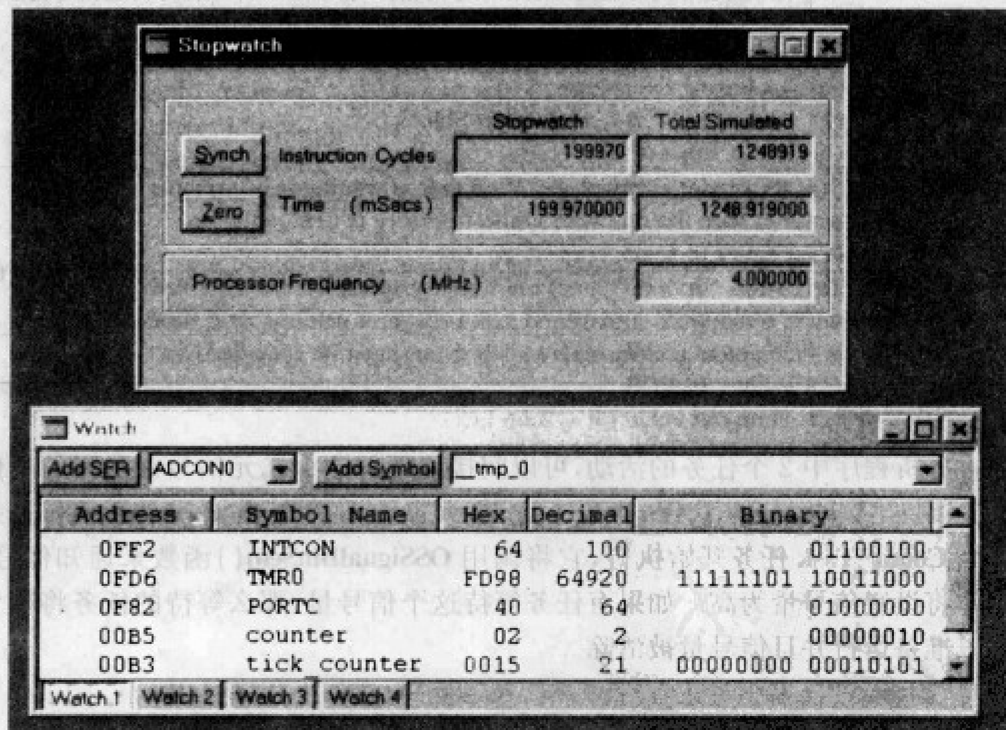


图 19-7 用于跟踪 RTOS 时钟滴答的 Watch 和 Stopwatch 窗口

使用仿真器中的调试命令来控制程序的运行。运行程序之后，它将停在遇到的第 1 个断点处。假设定时器的中断没有发生，那么这个断点将在 **Count_Task** 内。由于它是第 1 个被创建的任务，因此延迟不会起作用直到第 1 次 **OSDelay()** 迭代。在任务中，**counter** 的值被加 1。然后信号量被设置为高，再次运行程序并到达 **Display_Task** 内的断点。在这里，端口 C 的值被程序设置为 20_H。接着运行，程序将到达定时器 ISR 的断点处。

此时，清零 Stopwatch 窗口中的值。再次运行，程序将重复地停在定时器 ISR 处的断点。它们是时钟滴答。你可以发现程序每次停在断点时，Stopwatch 窗口中的递增

值都是 10ms 左右。有趣的是，递增的误差值不是常数并且依赖于其他程序的活动，而且这些活动与定时器是不同步的。这就验证了在 19.5.3 节中提到的时钟滴答近似精确的说法。在 20 个时钟滴答之后，由于 **Count_Task** 任务延时结束，它开始第 2 次运行。在设置了信号量之后，Display_Task 任务紧接着执行。可以从 Stopwatch 窗口中看到这 2 个任务都能够在一个时间片内完成。

图 19-8 以图例的形式表示了这个程序大概的行为方式。它显示了程序中总体的活动序列，当时图中没有画出精确的水平时间轴。

498

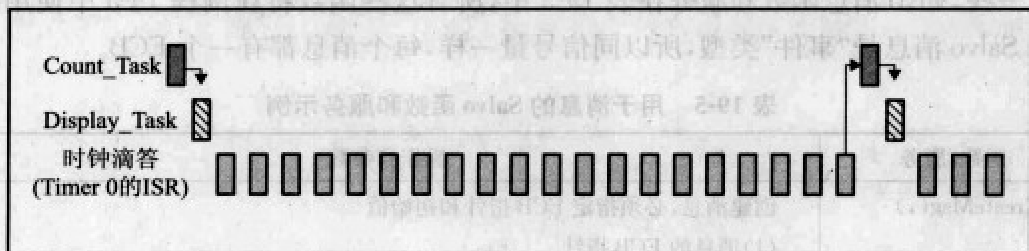


图 19-8　程序仿真时的断点发生情况

任务指南

有 2 张屏幕截图是在程序仿真时得到的，如图 19-7 所示。从图中可以推导出程序在所有停过的断点处的一些尽可能详细的信息。

19.5.7　运行程序

如果有 Derbot 硬件，你可以下载这个程序到硬件上实际运行起来。LED 将以令人满意的方式闪烁。当你看到 LED 闪烁，应该想到：每一个 LED 灯的状态改变都是由于时钟滴答累积到一定数量，从而导致一个任务被释放，使得一个计数器递增，再导致一个信号量的值改变，最后改变 LED 显示的。我们已经实现了一个很简单的程序，但是这个基本过程的简练和其实现的功能是非常令人满意的。对于大部分具有挑战性的应用，同样可以达到这种结构的简练和功能强大的统一。

19.6　使用 Salvo 消息和增加 RTOS 复杂度

下一步我们将学习 Salvo 中最后一个新资源——消息。但是消息的使用将导致程序的复杂度大大增加，我们将以更接近实际的方式来使用 RTOS 的特性。

消息提供了一种传统的方法来进行任务间的数据传递和活动同步。消息在某些特点上是类似于信号量的——它需要被创建；它可以从程序中某处获取数据，然后将数据再传递到其他位置，同时这可能会释放一个任务。

关于消息的术语有一些模糊。我们创建一个“消息”（即一个 Salvo 数据结构），然

499

后它就能携带任何数量的消息(即从程序一个地方传递到另一个地方的数据)。因此，我们称当前的 Salvo 数据结构为消息，那么把数据添加到消息的这个动作称为“发信号”给消息。

在 Salvo 中，消息携带的信号本身可以是任何数据类型：从字符到数组。消息传递的信息其实是指向信号的指针。在发信号给消息时由程序员来保证指针指向正确的数据并且在接收端能够被正确解析。要注意消息可以在程序的任何地方发出。比如由中断发出一个消息，然后由某个任务来接收这个消息。

一些 Salvo 消息函数和服务在表 19-5 中，所有这些函数将在例程 19-5 中使用到。由于 Salvo 消息是“事件”类型，所以同信号量一样，每个消息都有一个 ECB。

表 19-5　用于消息的 Salvo 函数和服务示例

函数/服务	操作和参数
OSCreateMsg(,)	创建消息，必须指定 ECB 指针和初始值 (1)消息的 ECB 指针 (2)消息指针
OSSignalMsg(,)	发信号给消息，即把数据添加到消息中。如果一个或者多个任务等待这个消息，那么最高优先级的任务变为就绪 (1)消息的 ECB 指针 (2)消息指针
OS_WaitMsg(,,,)*	任务一直等待(处于“等待”状态)直到消息被通知。此时，消息指针(参数 2)将指向被通知的消息，并且任务继续运行。如果暂停时间结束，任务也会继续运行。必须指定暂停时间 (1)消息的 ECB 指针 (2)消息指针 (3)暂停时间(以系统的时钟滴答为单位) (4)上下文切换符号
OSECB()	同表 19-4
OSNO_TIMEOUT	同表 19-4
OStypeMsgP	定义消息指针的数据类型。它是许多 Salvo 预定义数据类型之一，必须正确使用

* 引发上下文切换。

19.7　一个使用消息的例程

例程 19-5 是一个接近实际应用的基于 Salvo 的程序。它是在例程 15-3 中 Derbot AGV“盲目导航”程序的基础之上编写的，但是其中还使用到 8.6.5 节讲到的超声波传感器。传感器是面朝上安装在 Derbot AGV 上的，如图 19-9 所示。注意，由于传感器

同 LED 共用端口位，所以 LED 在程序中不可用。

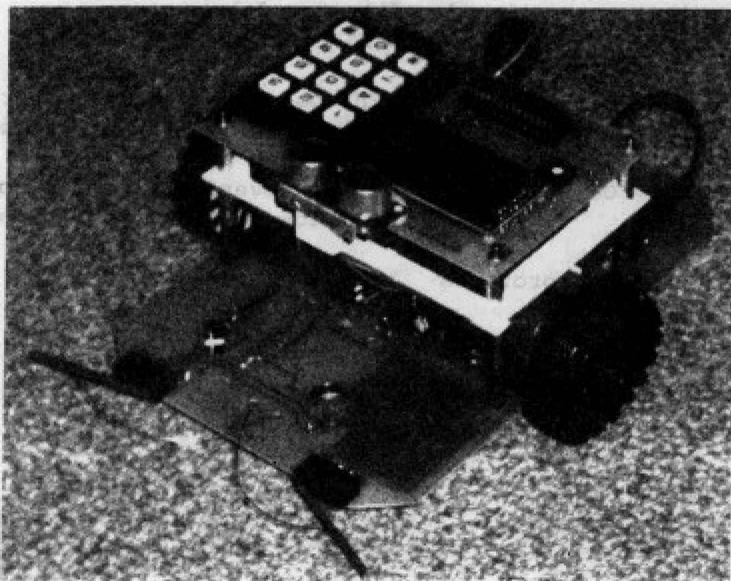


图 19-9 安装有面朝上的超声波传感器的 Derbot AGV

除了具备"盲目导航"的功能之外，当 Derbot AGV 行驶于悬垂物体下面时，比如在一个椅子下，程序还能够检测到上面的物体，然后旋转并离开这里。

现在，程序中有 2 个不同优先级的任务和 2 个不同优先级的中断。与以前的任务不同，现在每个任务都是一个较大的代码块并且包括多个上下文切换。程序中只定义了一个"消息"，使用它将消息信号从一个中断和其他任务传递到控制电机的任务。程序中多个地方使用到延迟。

除了创建的是一个消息而不是信号量之外，**main** 函数与以前例程中的非常类似。

例程 19-5 包含超声波悬垂检测功能的 Derbot AGV"盲目导航"程序


```
/**************************************************************************
rtos_ex3
Implements Derbot Blind Navigation, with upward-looking US sensor to
detect if AGV is going under an overhang.
Tasks are: Ultrasound Sensor (higher priority), Motor_set (lower priority).
Interrupts are: Microswitch (Low priority) & Timer 0 (High, for clock tick).
One message and numerous delays are also used.
Applies Salvo LITE with Library sfc18sfa. Can run on Derbot, or be simulated.
TJW 3.1.06                                                  Tested 4.1.06
**************************************************************************
//Clock is 4MHz

#include <salvo.h>
#undef OSC   //necessary for this Salvo version,as it also defines this name
#include <p18f242.h>
```

```
#include <timers.h>          //header file for delays
#include <delays.h>          //header file for delays
#include <pwm.h>             //header file for PWM

#pragma config OSC = HS, OSCS = OFF   //HS oscillator, oscillator switch off
#pragma config PWRT = ON, BOR = OFF   //pwr-up timer on, brown-out detect off
#pragma config WDT = OFF              //watchdog timer off
#pragma config STVR = ON, LVP = OFF   //Stack overflow reset enable on,
                                        //low voltage programming off

//User-defined function prototypes
      void Micro_Init(void);
      void leftmot_fwd (void);
      void rtmot_fwd (void);
      void leftmot_rev (void);
      void rtmot_rev (void);
```

Line 32

```
//These functions are tasks.
       void Motor_Task( void );  //Sets motor according to messages received
       void USnd_Task( void );   //Fires Ultrasound Sensor periodically

//Define labels for context switches
_OSLabel(Motor_Task1)
_OSLabel(Motor_Task2)
_OSLabel(Motor_Task3)
_OSLabel(Motor_Task4)
_OSLabel(Motor_Task5)
_OSLabel(USnd_Task1)
_OSLabel(USnd_Task2)
_OSLabel(USnd_Task3)
_OSLabel(LED_Task1)
_OSLabel(LED_Task2)

//Carries messages from microswitch and ultrasound
#define Msg_to_Motor OSECBP(1)

       char Hole = 0x18;   //This value used, but never tested
```

Line 53

```
/**************************************************************************
User-defined Functions, including RTOS Tasks.
**************************************************************************/
//This task controls motor action, determined by messages recd from elsewhere
void Motor_Task( void )
{      static char msge;        //hold message once recd
       OStypeMsgP msgP;         //Declare msgP as special Salvo pointer type
       for (;;)                 //set up the infinite Task loop
       {
       rtmot_fwd ();            //set motors running forward. This is status quo
       leftmot_fwd ();                          //until message arrives
       //Wait for message
       OS_WaitMsg(Msg_to_Motor,&msgP,OSNO_TIMEOUT,Motor_Task1);
```

```
// Line 67 Proceed when message arrives
      msge = *(char*)msgP;
      PORTAbits.RA5 = 0;  //stop motors for 500ms
      PORTAbits.RA2 = 0;
      OS_Delay (50,Motor_Task1);
      if( (msge == 0x80)| | (msge == 0x01) ) //was it a microswitch?
            {
            rtmot_rev ();                //Yes, so both motors reverse
            leftmot_rev ();
            OS_Delay (100,Motor_Task2);
            if(msge == 0x80)             //was left uswitch hit?
              {leftmot_fwd ();  //Yes, so turn right
              OS_Delay (80,Motor_Task3);
              }
              else             //right uswitch was hit
              {rtmot_fwd ();   //so turn left
              OS_Delay (80,Motor_Task4);
              }
            }
Line 86     else          //We're under an overhang, hence turn on spot
            {rtmot_rev ();
            leftmot_fwd ();
            OS_Delay (200,Motor_Task5);
            }
      }             //end of "for" loop
}
Line 93
/*Task periodically pulses Ultrasound, and sends a message if an overhang
detected. In this case, it suspends pulsing, to allow Derbot to exit*/
void USnd_Task(void)
{      int echo_time = 0;  //counts ultrasound distance measurement
    for (;;)        //set up the infinite Task loop
      {
      OS_Delay (20,USnd_Task1); //Task switch, and delay for 20x10ms,(200ms)
      OSDi();        //disable interrupts, this measurement is time sensitive
      echo_time = 0;
      PORTCbits.RC5 = 1;  //output us pulse.
      Delay10TCYx(2);     //20us delay approx, gives pulse width
      PORTCbits.RC5 = 0;
      Delay10TCYx(30);    //pause for op to be set high; ie blank for 5cm
Line 107
//Values in this loop are adjusted experimentally to give detection threshold
//of 30cm approx
      while (echo_time < 50)   //limit the measurement to close objects
            {Delay10TCYx(1);     //10us delay
            echo_time++;         //increment the counter
            if(PORTCbits.RC6 == 0) //send message if target detected
              {OSSignalMsg(Msg_to_Motor,(OStypeMsgP)&Hole);
              OSEi();       //enable interrupts before delay
              OS_Delay (250,USnd_Task2); //Suspend the USnd,
                                //to allow Derbot to exit "hole"
```

```
            OS_Delay (250,USnd_Task3);
            break;
            }
         }
      OSEi();                        //enable interrupts
      OS_Yield(USnd_Task2);
    }                                //end of "for" loop
}
```

// **Line 126** This function initialises the Microcontroller peripherals

```
void Micro_Init(void)
{
//Initialise Ports
       TRISA = 0b00000000; //All bits output, 2 & 5 used for motor enables.
       TRISB = 0b00110000; //Bits 5 and 4 (microswitches) only are input,
       TRISC = 0b11000000; //All bits output except 7 (mode switch) & 6
                                  //(USnd echo), 1 & 2 used for PWM
       ADCON1 = 0b00000110; //Set Port A for digital i/o
//Switch all outputs off
       PORTA = PORTB = PORTC = 0;
```

Line 137

```
/*Initialise Timer 0: interrupt enabled,16-bit operation, internal clock,
prescaler divide by 16, hence (with 4MHz clock) input cycle period of 16us*/
       OpenTimer0 (TIMER_INT_ON & T0_SOURCE_INT & T0_16BIT & T0_PS_1_16);
       WriteTimer0 (64918);                  //and initialise
//Initialise PWM
       OpenTimer2 (TIMER_INT_OFF & T2_PS_1_1 & T2_POST_1_1);
       OpenPWM1 (0xFF); //Enable PWM1 and set period
       OpenPWM2 (0xFF); //Enable PWM2 and set period
//Set Port B Interrupt on change to be low priority
       RCONbits.IPEN = 1; //Enable low priority interrupts
       INTCON2bits.RBIP = 0; //Set port change bit to be low priority
       INTCONbits.RBIE = 1;//Enable Port B change interrupt
}
```

Line 151

```
/*************************************************************************
Main
*************************************************************************/
void main( void )
{
//Initialise Microcontroller
       Micro_Init();
       Delay10KTCYx (250);          //pause 2.5secs with conventional delay
//Initialise RTOS
       OSInit();
//Create Tasks and Message
       OSCreateTask(Motor_Task, OSTCBP(1), 4);
       OSCreateTask(USnd_Task, OSTCBP(2), 2);
       OSCreateMsg(Msg_to_Motor, (OStypeMsgP)0);
//Enable Global interrupts
       OSEi();
```

```
//Scheduling Loop
    for (;;)
    OSSched();
}

/***************************************************************************
Motor Drive Functions
***************************************************************************/
...
(Same motor drive functions as Program Example 15.3.
```

19.7.1 选择库和配置

如果只考虑 Salvo 配置，那么这个程序与前面的程序相比没有什么不同。因为同前面一样，它也包含 2 个任务和一个事件。在这个例子中，事件就是指消息。因此，在这个项目中可以继续使用 **sfc18sfa.lib** 库，并且 **rtos_ex2** 项目中的 **salvocfg.h** 文件可以复制到当前项目中继续使用。

19.7.2 任务：USnd_Task

这个任务周期性地触发超声波传感器发出脉冲来检测在 AGV 上面是否有悬垂物。如果检测到悬垂物，它将发出一个消息，然后由 **Motor_Task** 来接收这个消息。由于对电机的控制依赖于这个任务的测量，因此 **USnd_Task** 任务相应地被设置为较高优先级。

USnd_Task 任务的结构是根据例程 9-7 编写的，它使用软件延时的方法首先产生一个脉冲，然后对回波进行计时。任务从程序清单的第 96 行开始。在任务代码中的无限循环开始处，调用了 **OS_Delay()** 函数使得任务每 20 个时钟滴答执行一次。当延时结束之后，通过设置端口 C 的位 5 为高，然后调用一个 20μs 的延时程序，之后再次设置位 5 为低，这将导致在端口 C 上输出了一个脉冲。

从程序第 110 行开始的循环是用于回波定时测量的，但是它是近似精确的并且仅用于短程测量。因此，在 **while** 循环中插入了一个有效的 50 个周期的延迟。如果回波（在端口 C 的位 6）在定时循环期间变低，它将产生一个消息。在这种情况下，任务被延迟一段很长的时间。在任务中，2 次调用了 **OS_Delay()** 函数，这是由于该函数携带的延时参数是 8 位宽的。在 AGV 转弯（由其他任务控制下）并离开悬垂物的过程中，超声波的动作由这段延时来抑制。但是如果回波不降为低，那么定时循环会超时，任务调用 **OS_Yield()** 函数主动让出 CPU 控制权。

19.7.3 任务：Motor_Task

Motor_Task 任务根据从端口 B 的中断和 **USnd_Task** 任务接收到的消息来控制电

机。任务是从程序清单中第 58 行开始的。变量 **msge** 在任务最开始处被声明，它用来存储发生的消息。然后程序中使用了 Salvo 中一个特殊的数据类型 **OSTypeMsgP**（表 505 19-5 中）定义了一个消息指针 **msgP**。

启动 2 个电机之后，任务调用 **OS_WaitMsg( )** 等待消息到达。根据接收的消息不同，任务运行的状态也不同。当电机设置成新状态之后，调用 **OS_Delay( )** 函数强制给任务加一个延时。由于调度器最先接收到消息，然后它决定任务是否应该被启用，所以在等待消息和延时期间，任务完全处于非激活态。

19.7.4 消息的用法

程序部分功能取决于消息的用法。关于消息用法方面的一些重要的程序行复制如下。不管怎么样，程序都是利用表 19-5 总结的 Salvo 函数。在程序中，只创建了一个 Salvo 消息 **Msg_to_Motor**，但是在程序中它可以从不同的地方携带不同内容的消息。**Motor_Task** 任务用来接收所有这些消息。

```
...
//Carries messages from microswitch and ultrasound
#define Msg_to_Motor OSECBP(1)
...
From main
OSCreateMsg(Msg_to_Motor, (OStypeMsgP)0);
...
From Motor_Task Function
static char msge;      //hold message once recd
OStypeMsgP msgP;       //Declare msgP as special Salvo pointer type
...
//Wait for message
OS_WaitMsg(Msg_to_Motor,&msgP,OSNO_TIMEOUT,Motor_Task1);
//Proceed when message arrives
msge = *(char*)msgP;
...
```

在程序清单靠前的位置（第 50 行）定义了一个消息 **Msg_to_Motor**。如同前面的信号量一样，它其实是一个指向 ECB 的指针。在这里，使用宏定义来命名这个消息，这是由于 3 个 Salvo 函数中使用这个宏定义来识别该消息。

在 **main** 函数中，调用 **OSCreateMsg( )** 来创建这个消息。程序在最初第 1 次调用 **Motor_Task( )** 之后，接着执行到 **OS_WaitMsg( )** 处。这将引发上下文切换并强制任务进入等待状态直到消息 **Msg_to_Motor** 被通知。函数携带的任务暂停时间参数没有显式指定，那么将使用 **OSNO_TIMEOUT** 这个默认值。**msgP** 指向发生的消息信号的指针，它是在 **Motor_Task( )** 函数的开始处被声明的。

消息是在 **uswitch_isr( )**（在下面所示的一个例子中）和 **USnd_Task( )** 中被通知的。消息通知的格式如下所示。预定义的名称 **Msg_to_Motor** 再次用来提供消息指针。消息的信号值 **Rt_usw** 在前面已经定义。函数的入口参数是指向消息信号的指针，它是 Salvo 特殊的 **OStypeMsgP** 类型的变量。

506

```
From uswitch_isr Function
...
char Rt_usw = 0x01;
...
if (PORTBbits.RB4 == 0)  //Test right uswitch
OSSignalMsg(Msg_to_Motor,(OStypeMsgP)&Rt_usw); //Send message
...
```

19.7.5 中断的使用和 ISR

本小节是本书仅有的一个使用 PIC 18 系列微控制器的 2 个高和低优先级中断的例子，所以这个程序的设置很值得我们学习。Timer 0 的中断用于产生周期为 10ms 的时钟滴答，正如例程 19-3 中的一样。这个例子还使用了端口 B 的电平变化中断功能，它用于检测微动开关是否被按下。

程序主要是在 **Micro_Init()** 函数结尾处配置中断的。其中使用到的寄存器可参考图 12-8 和图 12-10，**IPEN** 位的功能见图 12-7 所示。**IPEN** 位首先被设置为高（程序第 147 行），这将启用低优先级的中断通路。在程序下面 2 行，端口的电平变化中断被启用并且设置为低优先级。在 **main** 中通过调用 **OSEI()** 函数来启用全局中断。

例程 19-6 中包括了 2 个中断服务程序。微动开关的中断服务程序 **uswitch_isr()** 起始处有一个 8ms 的延时，它用于稳定开关反弹。之后，程序检测 2 个开关。开关为低将产生相应的消息。很可能这 2 个开关都为低，当开关被释放且处于激活态时程序进入 ISR，这种情况就会发生。最后，ISR 清除中断标志位。

例程 19-6　例程 19-5 的中断服务程序

```
/*****************************************************************************
ISR for rtos_ex3
There are two interrupt sources:
High Priority: Timer 0 for "clock tick",
Low Priority: microswitch (Port B change)for collision

TJW 3.1.06                                              Tested 5.1.06
*****************************************************************************/
#include <salvo.h>
#include <p18F242.h>
#include <timers.h>        //header file for timers
#include <delays.h>        //header file for delays

//function prototypes
void timer0_isr (void);
void uswitch_isr (void);

static unsigned int tick_counter = 0;

//These are values for messages
      char Rt_usw = 0x01;
      char Left_usw = 0x80;

//Carries messages from microswitch and ultrasound
#define Msg_to_Motor OSECBP(1)
```

507

```
/*****************************************************************************
Timer Interrupt (High Priority)
*****************************************************************************/

//Define the high priority interrupt vector to be at 0008h
#pragma code high_vector=0x08
void interrupt_at_high (void)
{
  _asm GOTO timer0_isr _endasm //jump to ISR
}

#pragma code //Return to default code section
//Function timer0_isr specified as high-priority ISR
#pragma interrupt timer0_isr

//timer0_isr function.
void timer0_isr (void)
{
      WriteTimer0 (64918);  //Timer reload value gives 625 cycles to
          //overflow, less compensation for interrupt latency
      OSTimer();
      tick_counter++;       //increment tick counter, (for simulation)
      INTCONbits.TMR0IF = 0; //Clear TMR0 interrupt flag
}
/*****************************************************************************
Microswitch Interrupt (Low Priority)
*****************************************************************************/

//Define the low priority interrupt vector to be at 0018h
#pragma code low_vector=0x18
void interrupt_at_low (void)
{
  _asm GOTO uswitch_isr _endasm  //jump to ISR
}
#pragma code                     //Return to default code section

//Function uswitch_isr specified as low-priority ISR
#pragma interruptlow uswitch_isr

//uswitch_isr function.
void uswitch_isr (void)
{
      Delay1KTCYx(8);             //8ms delay to ensure debounce
      if (PORTBbits.RB4 == 0)     //Test right uswitch
      OSSignalMsg(Msg_to_Motor,(OStypeMsgP)&Rt_usw); //Send message
      if (PORTBbits.RB5 == 0)     //Test left uswitch
      OSSignalMsg(Msg_to_Motor,(OStypeMsgP)&Left_usw); //Send message
//quite possible to land here with neither switch low any more, as interrupt
//will sense switch release
      INTCONbits.RBIF = 0;        //Clear Port B interrupt flag
}
```

19.7.6 仿真或者运行程序

如果你拥有一个 Derbot AGV，可以在上面运行这个有趣的程序。由于程序的运行依赖超声波传感器并且其中多处使用到了延时，所以不太方便使用软件来仿真这个程序。

19.8 RTOS 开销

本章旨在介绍使用 RTOS 进行编程方式的强大功能。但是我们必须认识到：Salvo Lite 上的应用开发只能利用受限的 Salvo 功能或者无法利用更多的 RTOS 特性。

在认识到这种编程方式具备强大功能的同时，了解它附带的代价也是很重要的。这些代价分为以下 3 类。

(1) 经济代价。一旦我们不再使用免费 RTOS，比如 Salvo Lite 或者类似的，就需要购买一个商业 RTOS 或者花时间(因此也是花钱)重新设计一个 RTOS。

(2) 程序大小代价。在 RTOS 上编写的程序必将占用更多存储空间。

(3) 执行时间代价。RTOS 操作系统本身的运行开销导致需要执行的代码量明显增加，因此程序运行会更慢。

这些代价类似于将程序从汇编语言转移到 C 语言需要付出代价。在这里，使用 RTOS 进行应用开发的这种编程方式功能更强大，但是随之也需要付出一些代价。上面总结的后 2 条代价决定了程序的实际性能。我们需要量化它们的影响，但是这并不容易。困难在于代码量大小和执行时间都依赖于当前程序的具体实现。比如，不能简单地说一个 RTOS 基于的程序占用的存储空间是传统顺序编程的 2 倍或者运行速度减半。有时候，人们使用基准程序在产生相同功能结果的不同编程实现之间进行比较。

Salvo 用户手册[19.1]中有一章(第 9 章)是关于“性能”的。这一章较为实用，它给出了 Salvo RTOS 和其组件的一些具体的性能特性。

就本书来说，我们有几个程序可以用于非正式的基准程序。比如例程 8-4 中使用汇编语言编写的 Derbot“盲目导航”程序和例程 15-3 中使用 C 语言编写的“盲目导航”程序，虽然它们使用的微控制器是不相同的。例程 17-3 中最初以传统 C 语言方式编写的一个基于中断的“闪烁 LED”程序和后来基于 RTOS 版本的例程 19-3。通过 **.map** 文件(见 17.10.3 节)来查看每对程序的内存使用率，你可以对它们作一个简单的比较。比较之后，你可能希望继续优化 C 语言方式的或者基于 RTOS 方式的程序。这 2 种方式都有可优化的余地，在 C 编译器和 Salvo 手册中对某些优化选项进行了描述。 509

小结

❑ 对于小型嵌入式环境来说，Salvo 是一个有效的实时操作系统；本章使用一些例子讲解了

Salvo 所有关键的 RTOS 特性，它非常适合于 PIC 环境。

- 使用 RTOS 进行应用开发导致产生了一种新的编程方式。其中，任务、优先级和事件是编程的关键。
- 使用 RTOS 编程也需要付出一些代价，包括经济、存储空间和执行时间这 3 个方面的。我们在决定是否使用 RTOS 之前，必须先了解和估算这些代价。

参考文献

19.1. Salvo™ – The RTOS that runs in tiny places™, User Manual, Version 3.2.2. Pumpkin Inc.; http://www.pumpkininc.com

19.2. Salvo Compiler Reference Manual – Microchip MPLAB-C18 (2005). Code RM-MCC18. Pumpkin Inc.; http://www.pumpkininc.com

19.3. Building a Salvo Application with Microchip's MPLAB-C18 C Compiler and MPLAB IDE v. 6 (2004). Pumpkin Inc., AN-25.2004 ; http://www.pumpkininc.com

第五部分　网络互连技术

本部分是本书的最后一章，探讨了网络互连技术，网络互连是当今嵌入式系统中的一个至关重要的领域。

第 20 章
互连与网络

尽管嵌入式系统中的众多重要的领域在前面的19章中都已经介绍过，但还有一些重要的领域几乎未提及过。其中有一个领域非常重要，因此在最后一章中将对它进行介绍。

如果你实现了 Derbot AGV 上的手动控制器板和其他 I^2C 设备，那么你就创建过一个小的网络了。这种方法在每个不同的系统或子系统之间需要进行通信的环境中都会用到。在家、车间、汽车、工厂以及整个世界中，所制造的所有有用的东西都组织成网络。它们之间通信的方式多种多样：不仅仅是通过电的方式，还通过光纤、红外线或无线电进行通信。

本章介绍有关网络互连的问题——用于创建数据链路的互连中介，以及数据实际在整个链路上格式化、移动和解释的方法。从本意上讲，本章介绍了某种通信方法和技术。我们会发现，就像居住在地球上的种类繁多的生物一样，网络机制的种类也非常多，每种网络机制用于满足于它所面对的特定的环境需要。

在本章中，你将学到：

❑ 构建网络的一些基本概念；
❑ 连接的方法，包括通过红外线和无线电的无线连接方法；
❑ 网络协议的种类；
❑ 在这些领域中 PIC® 微控制器是如何使用的。

上述的每个话题本身都是一个主要的研究领域，本章的目的并不是对它们进行完整的介绍，而仅仅介绍它们的大致内容以及进一步发展的思路。本章中没有列举设计实例。

20.1 网络互连概述

在很多情况下，我们都需要连接不同的系统或子系统。在家庭环境中，家电自动化逐渐成为现实。在这里，不同的家用电器和家用设备都连接在一起，例如通过互联网。在别处，也需要网络互连。现代的汽车中也包含了大量的嵌入式系统，所有嵌入式系统都从事非常特定的活动，但都互连在一起。在家庭或汽车环境中，互连需保持很长的时间并且要稳定。但是，其他一些网络或互连是暂时的，例如通过无线链路从个人备忘记事本中下载数据到笔记本电脑，该无线链路只在下载数据时存在。嵌入式系统设计者对所有这些互连都感兴趣，并且他们为此做了很多挑战性的工作。

网络互连的传统方法是使用电缆，使电信号由一个子系统传递到另一个子系统。我现在写作的计算机并不是最新型的计算机，内部包含了一团电缆，连接着鼠标、打印

机、扫描仪、互联网、扬声器和我所有的 PIC 开发工具。采用电缆连接并不是唯一的方法，还可以通过非物理链路来实现数据连接。链路连接最常用的方法是使用光或无线电。每种方法都有许多不同的形式。我们早就有了电视机远端控制，通过红外线来通信。也已经使用无线电通信很多年了；在计算机环境中，已经使用非物理链路，它能够非常有效地用于数据链接。

由于网络非常重要，实现网络远比仅仅实现连接要难得多，它需要实现很多功能。在复杂系统中，也必须深入地解决怎样将数据格式化和解释、怎样实现寻址，以及怎样实现纠错。所有这些都非常独立于物理互连本身。为了在网络中通过不同节点通信，就必须有非常清晰的规则指出如何建立或解释一个报文。我们已经学过了一些互连标准的定义，例如 I²C。这组规则称为协议(protocol)，该名字来源于外交或法律词汇。下面让我们学习有关协议的知识。

协议概述

对于大型的通过网络互连的系统来说，协议会非常复杂，在协议中定义了通信链路的各个方面。一些方面明显而另一些方面不太明显。为了为定义协议这一复杂过程提供帮助，国际标准化组织(International Organization for Standardization, ISO)制定了“协议的规则(protocol for protocols)”，叫做开放式系统互连(Open System Interconnect, OSI)模型，如图 20-1 所示。OSI 模型定义了从具体的物理层次的规则(定义了我们使用的连接器类型或需要的电压)到更加抽象的规则(定义了数据如何编码和如何实现纠错)。

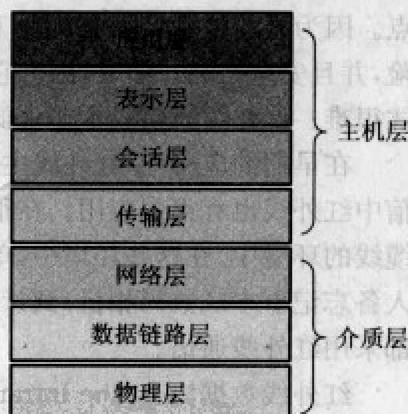


图 20-1 ISO 开放式系统互连模型

514

OSI 模型的每层为上层定义了一组服务，也就是每层都依赖于下层的服务。最低的 3 层依赖于网络本身，并且它们有时称作*介质层*(media layers)。物理层定义了物理和电气链路，例如，指定了所使用的连接器类型和如何用电信号表示数据。链路层用于提供可靠的数据流，并进行错误检测和纠正。网络层将数据放在网络的上下文环境中，并进行节点寻址。

OSI 模型的较高层都是用软件实现的。软件的实现是发生在主机上的，因此这些层有时叫作*主机层*(host layers)。软件实现通常称为*协议栈*(protocol stack)。对于一个给定的协议和硬件环境，它可作为一个标准的软件包来使用。使用协议栈的设计者可能需要与底层和顶层交互，在底层有物理互连，在顶层有同应用相连的软件接口。

该模型形成了一个框架，可以定义新的协议，并且在研究如今已使用的各种协议时提供一个有效的参考。在参考文献 20.1 中有更多的有关 ISO OSI 模型的信息。实际上，任何一种协议都不可能为 OSI 模型的每层提供完整的解决方案，或者说它只能

近似地指出每层的功能。

在本节后面将概述许多网络协议和连接方法。要实现这些协议，当然既需要一个物理系统，也需要一个软件系统。我们将会学到许多物理系统和软件系统，重点将放在现在已使用的硬件系统和软件驱动器上。

20.2 红外线连接

红外线数据通信已经出现很多年了，广泛地用于前面所提到的像电视机远端控制这样的应用中。使用低成本的半导体器件就能容易地产生和探测到红外线信号。由于红外线未在可见光谱内，所以它能非常方便地用于滤光片，过滤掉可见光，从而避免干涉。

所有 IR 链路的特征是数据通过一束调制光来通信。所以链路必须是瞄准线。链路距离通常很短，并且通信是一对一的。这样一个简单的特征也有大量的优点和缺点。因为光的传输具有方向性和局部性，几乎没有由于干涉造成数据传输错误的风险，并且安全性高。但是，因为它必须是瞄准线，所以在较大的网络中采用红外线的方式很难。红外线通信成本非常低，并且在使用上未受到法律的限制，这是一大优势。

在早期的应用中，红外线主要用在控制上，例如电视机远端控制。同样，在数据通信中红外线也有很多应用。在很多领域中都在使用红外线技术，特别是在只有单个电缆线的环境中，在这些环境中，单个电缆线被替换为红外线，例如从手持设备（例如个人备忘记事本或数码相机）到计算机的数据传输，或计算机和打印机之间的数据交互，都采用红外线通信。

红外线数据协会（the Infrared Data Association, IrDA）[20.2]由许多厂商组成，为 IR 链路（从简单控制到密集数据传输）定义一系列的标准。

IrDA 和 PIC 微控制器

对小型嵌入式系统来说，红外线通信有着广泛的应用。Microchip 公司提供了一些 IR 编码器/解码器 IC。其中一个是 MCP2122[20.3]，它的引脚连接框图如图 20-2 所示。左边的接口与微控制器相连，右边的接口与红外线源和接收器相连。这样，**TXIR** 引脚直接驱动一个 IR LED，而 **RXIR** 可以连接到一个传感器上。4 个引脚与主微控制器相连：**TX**、**RX**、**Reset** 和 **16x Clock**。**TX** 和 **RX** 线与主微控制器的 USART 相连，见 10.10 节。MCP2122 也需要 16 倍规定波特率的时钟源。该时钟源与 **16x Clock** 相连。它可用通过微控制器的 CCP 模块生成，见参考文献 20.4。主微控制器通过 **Reset** 输入将 IC 置为复位状态。

从前面的内容可以看到，PIC 微控制器的红外线端口的物理连接很简单，从这一级开始，在网络的两个节点之间可以开发非正式的链路。通常的应用是在一个 IrDA 标准下实现数据通信。如何实现不属于本书的讨论范围，但这些详细内容在许多 Micro-

515

chip 公司的应用手册中都可以见到，例如参考文献 20.5。

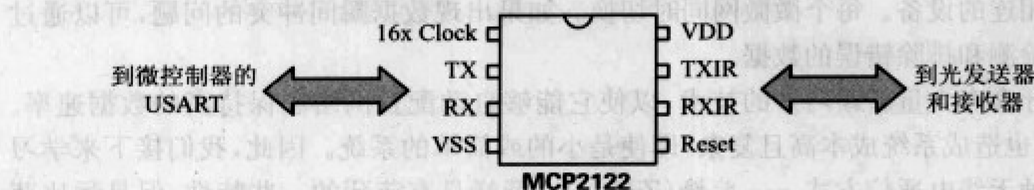


图 20-2 Microchip MCP2122 红外线编码器/解码器

20.3 无线电连接

虽然 IR 通信有许多明显的优势，但是它需要瞄准线通信，这是一个很大的缺点。所以，无线电链路受到青睐。低功耗的无线电系统可以实现局域连接并可以通过墙或其他（非导电的）障碍物进行通信。由于它是非瞄准线的，数据传输的一个巨大风险是企图占据同一空间的无线电网络之间的干涉。设想一个地方，例如饭店大厅，所有人都有无线电数据通信设备。我们是如何避免无线电之间大量干涉引起的风险的呢？本节简要介绍了一些无线电通信中使用的方法。

20.3.1 蓝牙

无线电数据传输领域中一个新的主要的传输方式是蓝牙（Bluetooth），它使无线电以一种有趣的方式来进行数据通信。蓝牙由一组电子厂商开发——蓝牙技术联盟（Bluetooth Special Interest Group），见参考文献 20.6。它工作在 2.402GHz～2.480GHz之间，最初这一带宽受到国际条约的保护，仅用于工业、科学和医疗（Industrial, Scientific and medical，ISM），但是如今也广泛地用于局域无线数据网络中。

蓝牙为诸如手机、计算机、数码相机或耳机之类的设备之间提供数据链接。它有如下特征：

- ☐ 低功耗的无线连接——功率大约为 1mW，相比之下，一部移动电话的功率为 3W；
- ☐ 数据传输的典型距离为 10m；
- ☐ 数据率起初为 1Mbit/s，如今为 3Mbit/s（Bluetooth 2.0）；
- ☐ 最多可同时连接 8 个设备；
- ☐ 采用了跳频展频（Spread-Spectrum Frequency Hopping，简写为 FHSS）技术，发送器以伪随机的方式每秒改变频率 1600 次。

516

当一个蓝牙设备检测到另一个蓝牙设备时，它们会自动地判断彼此是否需要建立连接，例如通过数据交换建立连接。这并不需要任何的人机交互。每个蓝牙设备都有自己的地址，并且通过地址可以判断它已经检测到的蓝牙设备是否正是所需要的。蓝牙系统按照这种方式相互通信，形成微微网（piconet）。一旦通信建立，微微网的各部

分同步跳频，因此它们保持通信。在单一的空间中可包含多个微微网，每个微微网含有彼此相连的设备。每个微微网同时切换。如果出现数据瞬间冲突的问题，可以通过软件来检测和排除错误的数据。

蓝牙含有大量新颖巧妙的技术，以使它能够自动配置网络和保持高的数据速率。然而，这也造成系统成本高且复杂，即使是小的或简单的系统。因此，我们接下来学习另外一种无线电通信方式——紫蜂(Zigbee)。紫蜂具有蓝牙的一些特性，但是远比蓝牙简单得多。

20.3.2 紫蜂

紫蜂是最近提出的一种无线电通信标准，由紫蜂联盟的成员制定和管理[20.7]。它具有蓝牙的特性，但是致力于比蓝牙更简单和更便宜，需要更小的软件开销。紫蜂采用 IEEE 802.15.4 的低速无线个人区域网络(Low-Rate Wireless Personal Network, LR-WPAN)标准。同蓝牙一样，它的工作频率范围为无线频谱的 ISM 带宽。

紫蜂特别适用于家庭自动化和其他测量控制系统，因为它使用简单小型的微控制器。数据率低且功耗小。

紫蜂节点有以下 3 种类型。

- 紫蜂协调器(Coordinator)——每个网络只有 1 个协调器，它是功能最强大的紫蜂节点，能够和其他网络通信并且保存自身网络的信息。
- 全功能设备(Full Function Device, FFD)——它可以从其他设备传递数据，因此能扮演路由的角色。
- 精简功能设备(Reduced Function Device, RFD)——这是最简单的设备。它只能与网络通信。

由于上述 3 种设备的功能逐一递减，那么它们所需要的存储器大小和运行时消耗的功率多少也是逐一递减的，同时它们的成本也是逐一递减的。

由于从节点大多时候处于休眠模式，所以整个网络可以达到最低功耗要求。从节点会短暂地苏醒，仅仅是确认它仍是网络的一部分。

20.3.3 紫蜂和 PIC 微控制器

紫蜂是一种新兴的无线电通信标准，一经推出，广受欢迎。在很多领域，它也广泛地同 PIC 微控制器一起使用。图 20-3 显示的是紫蜂节点的一种可能的物理实现方式。数据链路经由一个单芯片无线电收发器，例如 Chipcon CC2420[20.8]。微控制器通过 SPI 链路和某些控制线与无线电收发器交互。Microchip 公司在 PIC 微控制器中设计了一个协议栈，它可以用于使用紫蜂无线通信协议。在参考文献 20.9 中对此有所描述，并且该协议栈允许在网络中实现紫蜂协调器或精简功能设备。

517

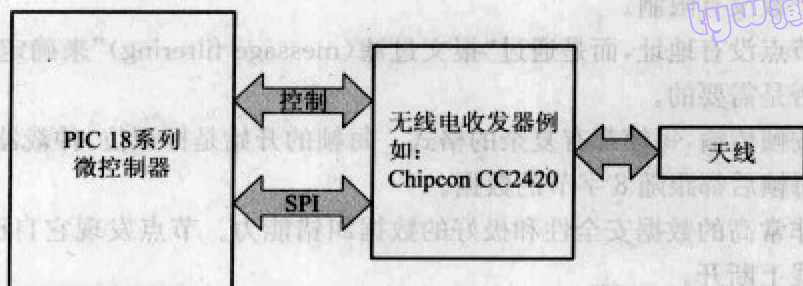


图 20-3 某种基于 PIC 的紫蜂实现

20.4 控制器局域网和局域互联网

在第 10 章中,我们学习了嵌入式环境中串行通信的 3 个主要工作方式——SPI、I^2C和异步。虽然它们都是非常好的串行通信标准,但每个都有自己的局限性。特别是,每个都不具有容错功能。本节将学到的 2 种串行标准是为非常特殊的应用开发的,高可靠性是应用中的一个关键需求。

20.4.1 控制器局域网

由于在汽车环境中数据通信的需求的增长,因而提出了控制器局域网(Controller Area Network,CAN)的概念。由于在汽车中电磁干涉较高并且温度和湿度范围较宽,因此对于任何信号和电子设备来说汽车环境确实是一个严酷的环境。此外,汽车需要非常高的可靠性。为家庭或办公室这样良好环境开发串行通信标准完全不适合汽车环境,因而需要开发一种新的标准。起初,CAN 由德国的博世公司开发。他们在 1991 年发布了 CAN 的 2.0 版本,并且其修改版本在 1993 年被 ISO 采纳为一种国际标准——ISO 11898。在本书编写时,2.0 版本可以从博世公司网站上下载[20.10]。

CAN 标准只能寻址图 20-1 中 ISO/OSI 模型的较低的 2 层,但在这么做时要使用一些非常革命性的方法。由于它有非常高的数据安全性,因而它相当复杂并且我们在这里不会对它进行详细地叙述。CAN 的主要特征如下所示。

518

- ❑ 异步、半双工通信方式,对于给定的系统有固定的比特率。最大比特率为 1Mbit/s。
- ❑ 配置是"对等的(peer to peer)",即所有节点一视同仁。但是有优先级机制。并不使用主从节点来对节点作标记。
- ❑ 总线上的逻辑值定义为"显式的(dominant)"或"隐式的(recessive)",显式逻辑值可以覆盖隐式逻辑值。否则无法定义物理互连。
- ❑ 总线访问灵活。由于所有节点都是对等的,任何一个节点都可以开始发出一条报文。如果出现同步访问的情况,使用一个特殊的仲裁进程进行处理,这不会浪费时间或数据。该仲裁进程根据优先级进行仲裁。

- □节点个数没有限制。
- □总线节点没有地址，而是通过"报文过滤(message filtering)"来确定总线上的数据是否是需要的。
- □数据按帧传输，每帧都有复杂的格式。每帧的开始是标识位，仲裁器由此进行仲裁。每帧后都跟随8字节的数据。
- □具有非常高的数据安全性和极好的数据纠错能力。节点发现它自己出错了，会从总线上断开。

现在CAN非常广泛地用于汽车环境中。图20-4显示的仅仅是假设的一部分汽车网络的框图。其中每个方框代表了一个小型的嵌入式系统——无线电、车门、座椅等。其中的车门我们在本书一开始就见过，见图1-2。所有子系统都通过CAN总线互连在一起。还有一个网络(图中未画出)也连接到中央控制模块。另一个网络控制着汽车的运动——例如，引擎、刹车和传动装置。

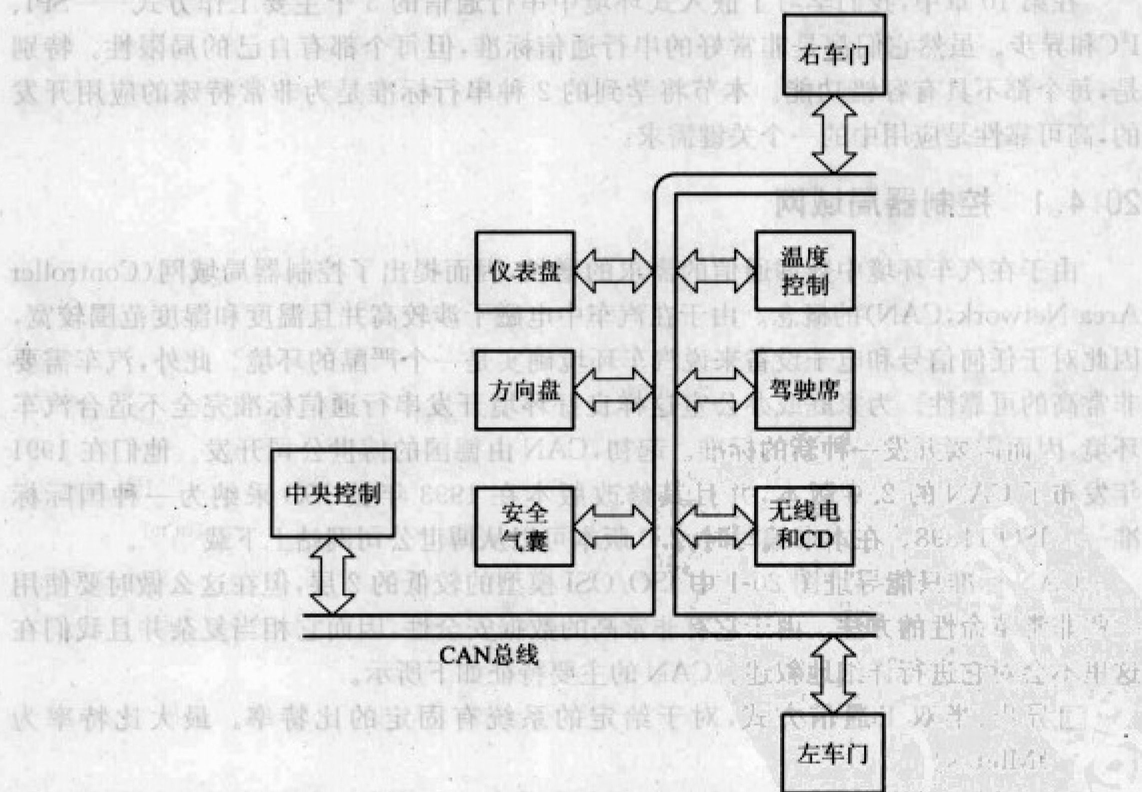


图20-4 部分车身控制网络

20.4.2 CAN和PIC微控制器

正如我们之前看见过带有I^2C或SPI端口的PIC微控制器，也有一些微控制器含有片上CAN模块。Microchip公司的CAN模块的当前版本称为ECAN™——增强型

CAN 模块，用来与 Microchip 公司早期的 CAN 相区别。例如 PIC 18F2480，在它的数据手册[20.11]中详细介绍了 ECAN 模块。

ECAN 模块复杂，具有的特征有：缓冲数据、按需格式化数据以及错误检测。ECAN 含有大量的控制寄存器。从零开始编写代码会非常费时。因此，Microchip 公司提供了一组 C 程序，见参考文献 20.12，开发人员可在构建程序时直接调用这些 C 程序。

无论使用哪种含有 ECAN 模块的微控制器，该微控制器都必须与物理总线相连，需要电气上的连接。通常这部分连接通过一个特殊的接口集成电路来完成，Microchip 公司的 MCP2551 就是这样一种接口 IC。图 20-5 是 MCP2551 的引脚连接框图，相关数据见参考文献 20.13。所使用的 CAN 总线与标准的 CAN 总线有些差异，它连接到 **CANH** 和 **CANL** 引脚上。一个外部电阻连接到 **Rs** 引脚上，用于控制数据的回转速率，可以使速率减慢以使电磁干涉减少到最小。微控制器从它的 ECAN 模块与 **RXD** 和 **TXD** 引脚相连。



图 20-5　MCP2551 CAN 收发器

20.4.3　局域互联网

虽然 CAN 能够提供非常高可靠性的数据通信，但是也正因为此 CAN 非常复杂并且成本很高。实际上，在汽车环境中并不是所有的链路都完全需要 CAN。因此，局域互联网（Local Interconnect Network，LIN）被开发出来同 CAN 一起使用。LIN 标准由 LIN 联盟制定和管理[20.14]。第 1 个版本在 1999 年发布，现在的 LIN 版本是 2.0 版。

LIN 总线确定为小型和低速的总线，主要同智能传感器和执行器通信。网络拓扑结构固定。LIN 总线中仅有一个主节点，其他节点均为从节点。因而主节点具有更强的处理能力。从节点仅需要非常有限的处理能力或仅仅是专用的硬件。因此，它们的成本很低。使成本最低的一种方式是从节点使用简单的 RC 振荡器，在数据交换时持续地进行同步。保守估计，最大的数据率为 20Kbit/s。主节点初始化所有的数据传输而仅有一个从节点会作出响应。当总线有多个访问时，没有任何机制处理这种情况。数据链路是单个线，如图 20-6 所示。同 CAN 总线一样，逻辑状态分为隐式（逻辑高）和显式（逻辑低）的。

520

一个 LIN 数据帧由帧头（header）和响应（response）组成。帧头总是来自于主节点，而数据响应总是来自于单个从节点。可选择的速度为 1kbaud～20kbaud。帧头由如下 3 个部分组成：

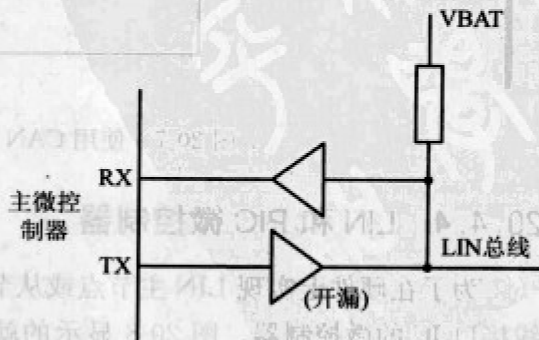


图 20-6　LIN 物理接口

- ❑ Break 域，告知从节点来了一条消息；
- ❑ Sync 字节，速率为预期的数据速率，包含的值为 55_H，从节点据此校准自己的时钟周期；
- ❑ 标识符(identifier)，标识是一个从节点还是多个从节点，以及指定要执行的动作。

响应是一条至多 8 字节的报文，只能由从节点放置在总线上。之后是校验和(Checksum)。在早期的总线版本中，只有数据字节才能被校验；在最近的版本中，标识符也可以被校验。

总线有 2 种状态——休眠(Sleep)和活跃(Active)。注意，在一段超时之后总线进入休眠状态，并且可以通过唤醒(Wakeup)帧来被重新激活。

有趣的是，LIN 标准也包括一些软件标准。LIN API(Application Programmers Interface，应用程序接口)提供了一组标准 C 语言函数，供程序员调用。在这些函数之间实现了所有的 LIN 功能。这使得软件开发变得容易并且易于测试。

在许多环境中，LIN 总线系统会与 CAN 总线系统相互交互，一同使用。图 20-7 显示的是一种可能的 LIN/CAN 系统。起中枢作用的是一个微控制器，例如 18F2480，该微控制器既具有 CAN 能力又具有 USART 能力。微控制器可作为 LIN 总线上的主节点和 CAN 总线上的对等节点。

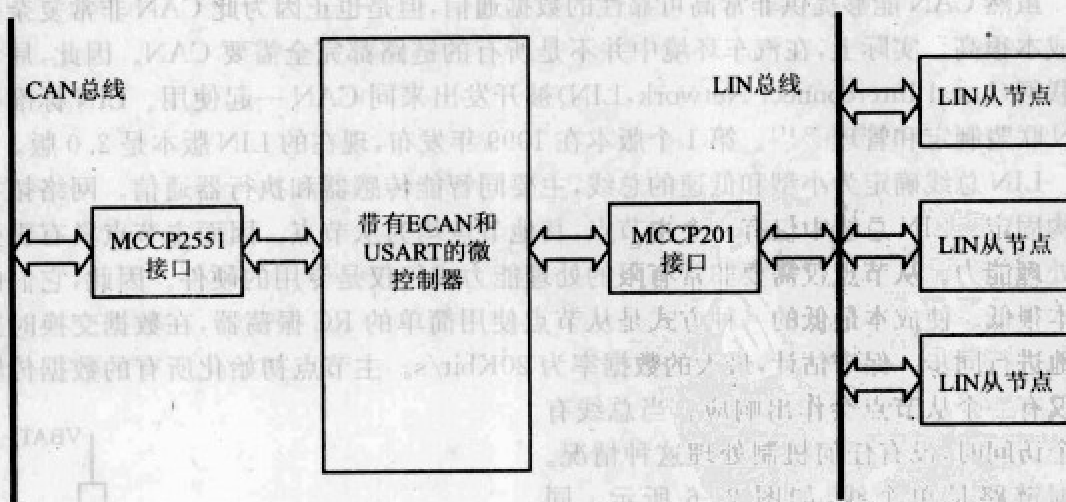


图 20-7 使用 CAN 和 LIN 总线的通信系统

20.4.4 LIN 和 PIC 微控制器

为了在硬件上实现 LIN 主节点或从节点，所需的仅是一个具有 USART 能力和总线接口 IC 的微控制器。图 20-8 显示的就是这样一个微控制器——Microchip 公司的 MCP201。它实现了图 20-6 中所示的缓冲。主微控制器的 USART 与 MCP201 的 **RXD** 和 **TXD** 引脚相连。电源通过 **VBAT** 引脚向 MCP201 供电。在板上有一个内部稳

521

压器，该接口芯片能够通过它的 **VREG** 引脚给主微控制器供电。Fault/SLPS 引脚在作为输出时用于标志错误，也可以设置数据偏斜。**LIN** 线与 LIN 总线相连。有关该设备的完整数据见参考文献 20.15。

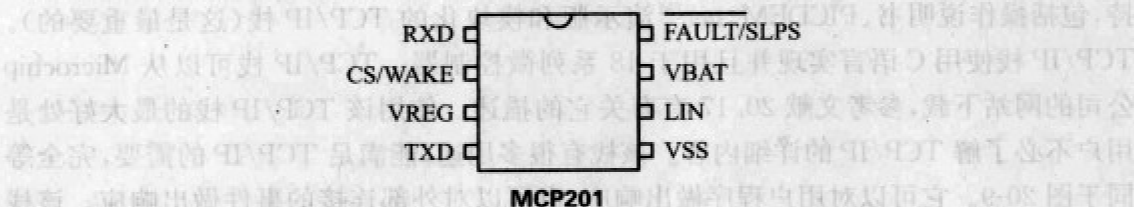


图 20-8 接口芯片实例——MCP201

同前面已经讨论的其他大多数标准一样，Microchip 公司已经发布了固件实例，方便代码的开发。参考文献 20.16 提供了最新(当前)2.0 总线版本下主节点和从节点的代码。

20.5 嵌入式系统和互联网

不言而喻，在过去的 10 年里，互联网改变了传统的通信方式，已成为全世界通信的主要方式。本书的重点是有关小型嵌入式系统的，而互联网通常被认为是用作连接最新台式计算机。这两者有共同点吗？回答是肯定的。可能这个回答会令人吃惊。将小型嵌入式系统连接到互联网是可能的，并且连接之后可以使通过网络互连的设备处于嵌入式控制之下。一旦链接建立，就可以做很多事情：监控状态、施加控制甚至下载程序或数据。有众多与互联网相连的嵌入式系统实例，例如，洗衣机可以通过互联网将一个亟待解决的问题告诉修理人员，自动售货机可以通过互联网通知总店机器内的商品已经卖完了，工厂可以通过互联网给已安装的防盗警铃下载最新版的程序，房主可以通过互联网从办公室打开家里的烤箱或者是检查车库门是否关上。

522

互联网通信使用大量的协议，通常这些协议集中起来统称为 TCP/IP，因为 TCP/IP 包含 2 个重要的通信协议——TCP(Transmission Control Protocol，传输控制协议)和 IP(Internet Protocol，网际协议)。图 20-9 显示的是一个网际协议栈的例子。图中也表明了它与 ISO/OSI 模型的关系。

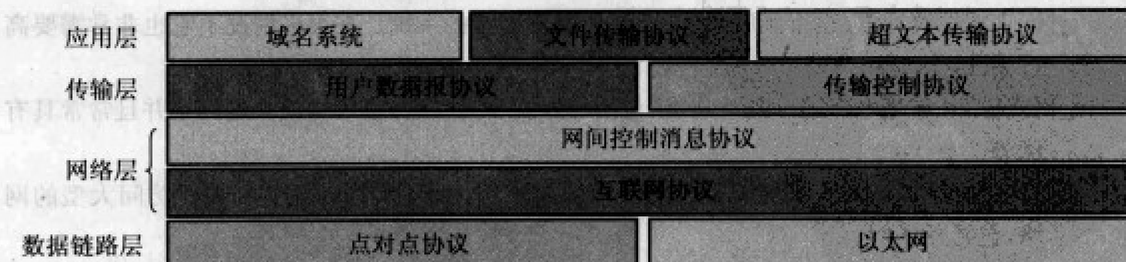


图 20-9 互联网协议栈

PIC 微控制器与互联网相连

网际协议是本章所见的最复杂的协议。但是，Microchip 公司对此提供了很好的支持，包括操作说明书、PICDEM. net™ 演示版和模块化的 TCP/IP 栈（这是最重要的）。TCP/IP 栈使用 C 语言实现并且用于 18 系列微控制器。TCP/IP 栈可以从 Microchip 公司的网站下载，参考文献 20.17 有有关它的描述。使用该 TCP/IP 栈的最大好处是用户不必了解 TCP/IP 的详细内容。该栈有很多用途，能满足 TCP/IP 的需要，完全等同于图 20-9。它可以对用户程序做出响应，也可以对外部连接的事件做出响应。该栈占据大约 20KB 的代码空间。

为了便于掌握如何将 PIC 微控制器连接到互联网，推荐使用 PICDEM. net 和与之配套的文档。

20.6 总结

在前面的 19 章中，我们已经学习了大量的 PIC 微控制器嵌入式系统的知识。从无到有，逐渐地为大家勾画出了一幅复杂而完整的微控制器结构图。我们使用汇编语言和 C 语言为微控制器编程，将微控制器与许多传感器和执行器相连，以及将微控制器与另一个微控制器相连形成一个微型网络。然后我们继续学习了如何将程序放在一个实时操作系统中。我们成功地通过一个电量适中的电池为微控制器供电。所有这些都表示了一个巨大的成就，如果你已经完全掌握了它们，就已经在使用微控制器进行嵌入式系统设计上做得很好了。最后一章指出了微控制器未来可能的方向。

523

希望你喜欢本书所提供的有关嵌入式系统方面的知识，也希望你能够继续享受通过嵌入式系统设计、构建和编写更多你自己思考的系统所带来的乐趣！

小结

- ❑ 现代系统对通信和网络有着强烈的需求。选择什么特征的网络受实际应用的驱动。灵活性、可靠性、易用性、数据速度、功耗和成本都是非常重要的考虑因素。
- ❑ 嵌入式系统通常（也不绝对）是低速和低数据容量的系统。在很多情况下它也非常需要高可靠性。
- ❑ 传统的互连是电互连。其他技术（包括红外线和无线电）也可构成互连网络并且常常具有优势。
- ❑ 有些情况下，嵌入式系统需要访问小型的局域网，而在其他情况下则需要访问大型的网络，包括互联网。
- ❑ 小型网络可以是专有的和固定的，例如 LIN 系统；也可以是灵活的和暂时的，例如蓝牙系统或紫蜂系统。
- ❑ CAN 网络具有非常高的可靠性，同时也更复杂。

□ Microchip 公司为实现很广范围的网络互连提供了非常有价值的支持，包括带有专有通信模块的微控制器、推荐电路和免费发布的固件。

参考文献

20.1. Green, D. C. (1995). *Data Communication*, 2nd edn. Longman. ISBN 0-582-24520-6.
20.2. Infrared Data Association (IrDA) 网址：http://www.irda.org/
20.3. MCP2122 Infrared Encoder/Decoder Data Sheet. (2004). Microchip Technology, DS21894B.
20.4. Interfacing the MCP2122 to the Host Controller (2004). Microchip Technology, Application Note AN946, DS00946A.
20.5. Programming the Pocket PC OS for Embedded IR Applications (2004). Microchip Technology, DS00926A.
20.6. 蓝牙技术官方网址： http://www.bluetooth.com/
20.7. 紫蜂联盟网址：http://www.zigbee.org/
20.8. CC2420 2.4 GHz IEEE 802.15.4/ZigBee-ready RF Transceiver Data Sheet（未更新）. Chipcon, SWRS041; http://www.chipcon.com/
20.9. Microchip Stack for the ZigBee™ Protocol (2004). Microchip Technology, AN965, DS00965A.
20.10. Bosch的CAN部分网址： www.can.bosch.com/
20.11. PIC18F2480/2580/4480/4580 Data Sheet (2003). Microchip Technology, DS21667D.
20.12. PIC18C ECAN 'C' Routines (2003). Microchip Technology, AN878, DS00878A.
20.13. MCP2551 High-Speed CAN Transceiver Data Sheet (2003). Microchip Technology, DS21667D.
20.14. LIN联盟网址：http://www.lin-subbus.org/
20.15. MCP201 LIN Transceiver with Voltage Regulator (2003). Microchip Technology, DS21730E.
20.16. LIN 2.0 Compliant Driver Using the PIC18XXXX Family Microcontrollers (2003). Microchip Technology, AN1009, DS01009A.
20.17. The Microchip TCP/IP Stack (2002). Microchip Technology, AN833, DS00833B.

524
525

附录 1　PIC®16 系列指令集

表 A1-1　PIC 16 系列指令集概览

助记符,操作数		描　述	周期	14 位操作码 MSB			LSB	受影响的状态	注
针对字节的文件寄存器操作									
ADDWF	f,d	将 W 和 f 相加	1	00	0111	dfff	ffff	C,DC,Z	1,2
ANDWF	f,d	W 和 f 相与	1	00	0101	dfff	ffff	Z	1,2
CLRF	f	f 清零	1	00	0001	lfff	ffff	Z	2
CLRW	—	W 清零	1	00	0001	0xxx	xxxx	Z	
COMF	f,d	f 求补	1	00	1001	dfff	ffff	Z	1,2
DECF	f,d	f 减 1	1	00	0011	dfff	ffff	Z	1,2
DECFSZ	f,d	f 减 1,为 0 则跳过	1(2)	00	1011	dfff	ffff		1,2,3
INCF	f,d	f 加 1	1	00	1010	dfff	ffff	Z	1,2
INCFSZ	f,d	f 加 1,为 0 则跳过	1(2)	00	1111	dfff	ffff		1,2,3
IORWF	f,d	W 和 f 同或	1	00	0100	dfff	ffff	Z	1,2
MOVF	f,d	移动 f	1	00	1000	dfff	ffff	Z	1,2
MOVWF	f	将 W 送至 f	1	00	0000	lfff	ffff		
NOP	—	空操作	1	00	0000	0xx0	0000		
RLF	f,d	f 带进位的循环左移	1	00	1101	dfff	ffff	C	1,2
RRF	f,d	f 带进位的循环右移	1	00	1100	dfff	ffff	C	1,2
SUBWF	f,d	f 减去 W	1	00	0010	dfff	ffff	C,DC,Z	1,2
SWAPF	f,d	f 半字节交换	1	00	1110	dfff	ffff		1,2
XORWF	f,d	W 和 f 异或	1	00	0110	dfff	ffff	Z	1,2
针对位的文件寄存器操作									
BCF	f,b	f 的位 b 清零	1	01	00bb	bfff	ffff		1,2
BSF	f,b	f 的位 b 置位	1	01	01bb	bfff	ffff		1,2
BTFSC	f,b	检测 f 的位 b,为 0 则跳过	1(2)	01	10bb	bfff	ffff		3
BTFSS	f,b	检测 f 的位 b,为 1 则跳过	1(2)	01	11bb	bfff	ffff		3
立即数与控制操作									
ADDLW	k	立即数和 W 相加	1	11	111x	kkkk	kkkk	C,DC,Z	
ANDLW	k	立即数和 W 相与	1	11	1001	kkkk	kkkk	Z	
CALL	k	调用子程序	2	10	0kkk	kkkk	kkkk		
CLRWDT	—	看门狗定时器清零	1	00	0000	0110	0100	$\overline{TO}$,$\overline{PD}$	
GOTO	k	跳转	2	10	1kkk	kkkk	kkkk		
IORLW	k	立即数与 W 同或	1	11	1000	kkkk	kkkk	Z	
MOVLW	k	立即数送到 W	1	11	00xx	kkkk	kkkk		
RETFIE	—	中断返回	2	00	0000	0000	1001		
RETLW	k	立即数送到 W,子程序返回	2	11	01xx	kkkk	kkkk		

（续）

助记符，操作数		描 述	周期	14 位操作码 MSb			LSb	受影响的状态	注
立即数与控制操作									
RETURN	—	子程序返回	2	00	0000	0000	1000		
SLEEP	—	进入休眠模式	1	00	0000	0110	0011	$\overline{TO}$,$\overline{PD}$	
SUBLW	k	立即数减去 W	1	11	110x	kkkk	kkkk	C,DC,Z	
XORLW	k	立即数和 W 异或	1	11	1010	kkkk	kkkk	Z	

注：1. 当 I/O 口寄存器用自身内容修改自身时（例如：MOVF PORTB，1），使用的值将是出现在引脚上的值，而非锁存器中的值。例如，如果一引脚配置为输入，其数据锁存器中的值为“1”，但此时外部器件将该引脚拉为低电平，则被写回数据锁存器的数据值将是“0”。

2. 如果预分频器被分配给 Timer 0 模块，那么对 TMR0 寄存器执行这条指令（适当时，d=1）将清零预分频器。

3. 如果程序计数器（PC）被修改或者执行一个条件检测为真的跳转，那么指令需要 2 个指令周期。第 2 个指令周期执行一条空操作 NOP 指令。

附录 2　电子乒乓球游戏

这个小型嵌入式系统游戏需要 2 位选手参加，每位选手有一个称为“球拍”的按钮。两位选手都可通过按下按钮来开始游戏。“乒乓球”用一行共 8 个发光二极管表示，游戏开始时，球会迅速地朝对方选手飞去，对方选手必须恰好在球位于最后一个发光二极管时按下他的按钮，而不能在其他位置按下。如果操作正确，球就会返回，游戏继续直到有一方违反游戏规则。一旦有一方违反了游戏规则，与没有违反游戏规则的选手的分数相关的发光二极管就会点亮。当球出局时，“出局”发光二极管会被点亮。

图 1-3 为电子乒乓球游戏的照片。图 A2-1 是电子乒乓球游戏的电路框图。本附录后面列出了电子乒乓球游戏的程序。

图 A2-1　电子乒乓球游戏的电路框图

例程 A2-1 电子乒乓球游戏的程序

```
;****************************************************************
;ELECTRONIC PING-PONG!
;This program drives the electronic ping-pong game,
;fixed speed, single mode of play.
;TJW 21.6.01
;****************************************************************
;
;Clock freq 800kHz approx (RC osc.)
;Port A 4     right paddle (ip)
;        3     left paddle (ip)
;        2     "out of play" led (op)
;        1     "Score Left" led (op)
;        0     "Score Right" led (op)
;Port B 7-0   "play" leds (all op)
;
;Config Word:        RC oscillator, WDT off,
;             power-up timer on, code protect off
;No Interrupts used
;
        list   p=16F84A
;specify SFRs
status   equ          03
porta    equu         05
trisa    equ          05
portb    equ          06
trisb    equ          06
;
;specify a constant
led_durn equ          20   ;no. of time inner loop is iterated, hence
                                ;time duration each led is lit.
;
;specify RAM locations
delcntr1 equ  10     ;used in 5ms delay SR
delcntr2 equ  11     ;used in 500ms delay SR
led_posn equ  12     ;holds current ball led posn.
loop_cntr equ 13     ;preloaded with value led_durn for every
                          ;led illumination, and counts down to 0 before
                          ;ball moves on
;
        org    00
        goto   start
;
; **"Initialise" State**
; Initialise
       org     0010
start bsf      status,5 ;select memory bank 1
       movlw   B'00011000'
       movwf   trisa   ;port A according to above pattern
       movlw   00
       movwf   trisb   ;all port B bits op
       bcf     status,5 ;select bank 0
;
```

```
;***Wait*** State
;set up initial led patterns
wait  movlw  04
      movwf  porta  ;switch on "out of play" led
      movlw  00
      movwf  portb  ;all play leds off
;
;check that both paddles are clear before allowing play to commence
      btfss  porta,4 ;right paddle pressed?
      goto   wait    ;yes, so wait
      btfss  porta,3 ;left paddle pressed?
      goto   wait    ;yes, so wait
;
;now ready for action, now wait until paddle pressed
wait1 btfss  porta,4 ;right paddle pressed?
      goto   r_to_l  ;yes, so start play
      btfss  porta,3 ;left paddle pressed?
      goto   l_to_r  ;yes, so start play
      goto   wait1
;
;**"Right-to-Left" State**
;play has started
r_to_l movlw 00      ;switch off "out of play"
      movwf  porta
      movlw  80      ;define ball start posn.
      movwf  led_posn
;loop to here every time led is to change
rtl_0 movlw  led_durn
      movwf  loop_cntr    ;preset length of led illumination
      movf   led_posn,w   ;output new ball posn
      movwf  portb
;loop to here n times for every led, where n = led_durn.
;Check for rule violations. Special conditions apply if
;ball is at start or end.
rtl_1 btfss  led_posn,7  ;is ball at start (ie posn 7)?
      goto   rtl_2  ;no, so move on
      ;yes, it's OK if right paddle still pressed, so don't test
      btfss  porta,3      ;left paddle pressed?
      goto   rt_myscore  ;yes, so score
      goto   rtlend
rtl_2 btfss  led_posn,0  ;is ball at end (ie posn 0)?
      goto   rtl_3  ;no, so move on
;here if ball at end, left paddle can force direction change
      btfss  porta,3       ;left paddle pressed?
      goto   l_to_r ;yes, so change direction - **state exit**
      btfss  porta,4      ;right paddle pressed?
      goto   rt_yrscore  ;yes, so left scores
      goto   rtlend
;here if neither start nor end posn.
rtl_3 btfss  porta,4      ;right paddle pressed?
      goto   score_left  ; yes, so score
      btfss  porta,3      ;left paddle pressed?
      goto   rt_myscore  ;yes, so score
```

```
;at then end of each loop call a delay
rtlend call  delay5
       decfsz loop_cntr   ;decrement loop counter, check if led is to move
       goto   rtl1
;here if ball moving on
       bcf    status,0
       rrf    led_posn,1
       btfsc  status,0     ;ball off end?
       goto   rt_myscore  ;yes, right's point
       goto   rtl0
;**state exit**
rt_myscore   goto score_right
rt_yrscore   goto score_left
;
;***"Left-to-Right" State**
l_to_r movlw 00            ;switch off "out of play"
       movwf porta
       movlw 01            ;define ball start posn.
       movwf led_posn
ltr_0 movlw  led_durn
       movwf  loop_cntr  ;determine length of led illumination
;go round this loop "duration" times, for every ball position
ltr_1 movf   led_posn,w ;output new ball posn
       movwf  portb
       btfss  led_posn,0 ;is ball at start (ie posn 0)?
       goto   ltr_2      ;no, so move on
       ;yes, OK if left paddle still pressed (so only test rt paddle)
       btfss  porta,4     ;right paddle pressed?
       goto   lft_myscore  ;yes, so score
       goto   ltrend
ltr_2 btfss  led_posn,7 ;is ball at end (ie posn 7)?
       goto   ltr_3      ;no, so move on
;here if ball at end,right paddle will change dirn,score right if left paddle
       btfss  porta,4     ;right paddle pressed?
       goto   r_to_l      ;yes, so change direction
       btfss  porta,3     ;left paddle pressed?
       goto   lft_yrscore ;yes, so right score
       goto   ltrend
;here if neither start nor end posn.
ltr_3 btfss porta,4      ;right paddle pressed?
       goto   lft_myscore ;yes, so score
       btfss  porta,3     ;left paddle pressed?
       goto   lft_yrscore ;yes, so score
ltrend call  delay5
       decfsz loop_cntr   ;decrement loop counter, check if led is to move
       goto   ltr_1
;here if ball moving on
       bcf    status,0     ;Clear Carry, as rlf rotates through it
       rlf    led_posn,1
       btfsc  status,0  ;ball off end?
       goto   lft_myscore ;yes, left's point
       goto   ltr_0
```

```
;**state exit**
lft_myscore goto score_left
lft_yrscore goto score_right
;
;***Score* State**
;here if Left has scored
score_left
      movlw  00
      movwf  portb  ;all play leds off
      bsf    porta,1
      call   delay500
      bcf    porta,1
      goto   wait
;here if Right has scored
score_right
      movlw  00
      movwf  portb  ;all play leds off
      bsf    porta,0
      call   delay500
      bcf    porta,0
      goto   wait
;
;********************************************
;SUBROUTINES
;********************************************
;Delay of 5ms approx. Instruction cycle time is 5us.
delay5  movlw D'200' ;200 cycles called,
                              ;each taking 5x5=25us
      movwf  delcntr1
del1  nop                ;5 inst cycles in this loop
      nop
      decfsz delcntr1,1
      goto   del1
      return
;
; Delay of 500ms (approx) - 100 calls to delay5
delay500 movlw D'100'
      movwf   delcntr2
del2  call    delay5
      decfsz  delcntr2,1
      goto    del2
      return
;
      end
```

附录 3 Derbot AGV 硬件设计细节

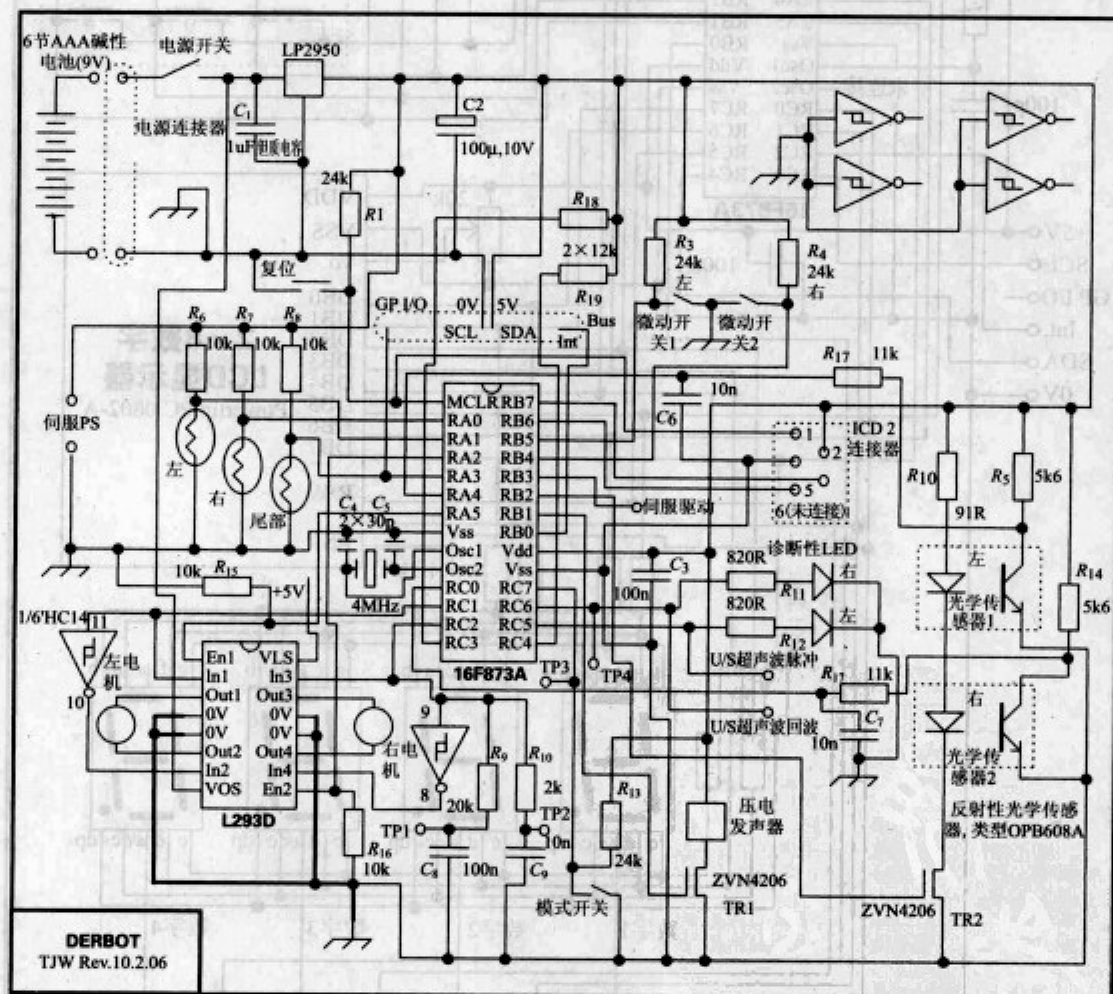


图 A3-1 Derbot 电路图



Derbot手动控制器
Dual LED/LCD TW Rev.22.6.05
注意：需要安装LCD或者LED显示器

7×22k

键盘

1 2 3 4 5 6 7 8 9 * 0 #

MCLR RB7 RA0 RB6 RA1 RB5 RA2 RB4 RA3 RB3 RA4 RB2 RA5 RB1 Vss RB0 Osc1 Vdd Osc2 Vss RC0 RC7 RC1 RC6 RC2 RC5 RC3 RC4

16F873A

8.2k

100p

未连接

+5V
SCL
GP I/O
Int.
SDA
0V

100n

20k

VDD VSS Vo DB0 DB1 DB2 DB3 DB4 DB5 DB6 DB7 R/W E RS

2×8数字
LCD显示器
Powertip PC 0802-A

4×1.2k

4×1.2k

g f a/ca b

e d a/dc dp

数字1 数字2 数字3 数字4

ZVN4206 ZVN4206 ZVN4206 ZVN4206

图 A3-2 Derbot“手动控制器”电路图

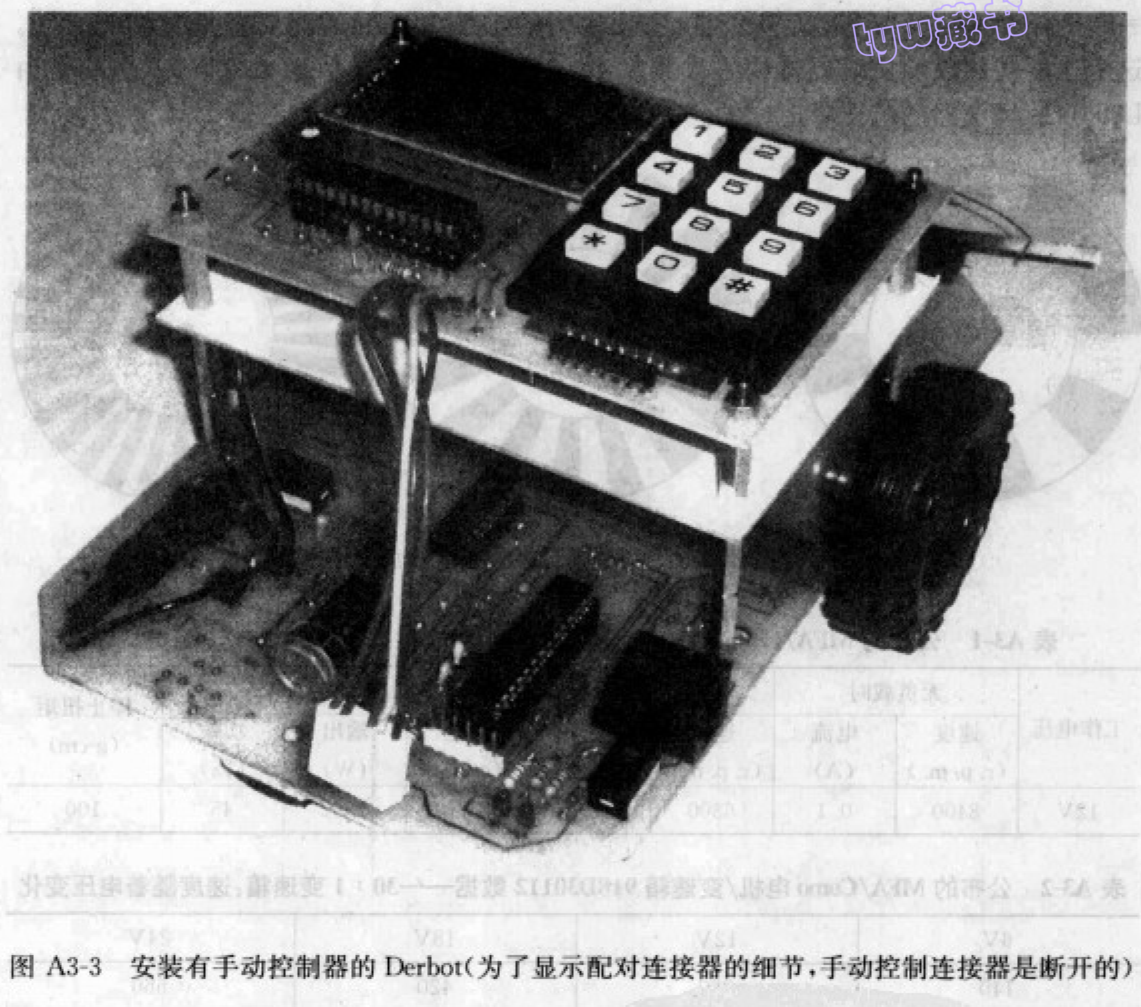


图 A3-3 安装有手动控制器的 Derbot(为了显示配对连接器的细节,手动控制连接器是断开的)

AGV 的增量轴角编码器

第 8 章讲述了如何使用光学传感器来制作一个简单的增量轴角编码器。下图显示了 3 种轴角图样:8、16 和 32 个黑/白周期。这 3 种均可以使用,但是对于 32 个周期的轴角图样,必须非常小心地控制轴角图样与传感器之间的距离。这 3 种轴角编码器使用的车轮直径都是 56.0mm,即周长为 176.0mm。因此,轴角编码器向前行驶的路程分辨率分别为 176.0/8=22.0mm、176.0/16=11.0mm、176.0/32=5.5mm。进一步说,如果车轮旋转速度为每分钟 n 转并且使用的是 16 个周期的轴角图样,那么轴角编码器的频率为 $16n/60$。

表 A3-1 和表 A3-2 给出了 Derbot 中使用的电机相关的参考数据[8,8],从中可以看到当电机电源为 9V 时,其速度为每分钟 210 转。因此,这是 Derbot 可以达到的最大速度。当使用 16 个周期的轴角图样时,相应产生的轴角编码器最大频率为 16×210/60 535

=56Hz。实际上，当电源为 9V 时，Derbot 中的空转电机的速度只有每分钟 154 转。这个速度转化成轴角编码器的最大频率为 41Hz。这个速度小于参考值，这主要是由 L293D 驱动电路的电压降造成的。

图 A3-4　8 个、16 个和 32 个周期的增量轴角编码器图样

表 A3-1　公布的 MFA/Como 电机/变速箱 918D30112 数据——RE280/1 电机特性

工作电压	无负载时		最大功效时					停止扭矩 (g-cm)
	速度 (r. p. m.)	电流 (A)	速度 (r. p. m.)	电流 (A)	扭矩 (g-cm)	输出 (W)	功效 (%)	
12V	8400	0.1	6300	0.3	25.0	1.62	45	100

表 A3-2　公布的 MFA/Como 电机/变速箱 918D30112 数据——30：1 变速箱：速度随着电压变化

6V	12V	18V	24V
140	280	420	560

附录 4　自主导向车的一些基本知识

该附录介绍了一些 AGV 设计方面的特定内容，它对于我们设计 Derbot 项目是必要的。

运动和车轮设计

在考虑小型 AGV 的运动装置之前，先观察一下我们周围一些带轮的交通工具。这里顺便提一下，车轮当然不是运动的唯一方式——还有各种各样轨道和步行方式的运动装置，但是它们不是太复杂，就是效率低，所以都不适合小型的 AGV 设计。最普通的车轮设计当然是电动车。一般地，它后面有 2 个驱动轮，前面是 2 个方向轮。这种设计具有很高的稳定性，很适合用于载人的交通工具。它也能够提供适度的（虽然不是很灵活）可操纵性——当将车停在狭窄的空间时，我们是否还记得车是很难控制的。

从许多可行的车轮配置（见参考文献 A4.1）中，我们选择了一种非常流行并且适合 AGV 的车轮。运动装置由 3 点支撑，其中 2 个点是 2 个独立驱动的车轮，第 3 个点是摩擦力小的滚轮或者滑轮。后者可以由一个小瓶来替代，但是这样会影响车的可操纵性，因此应该尽量避免使用它。车的重心正好落在后轮上。变量 A 表示 2 个车轮中心线之间的距离。

这种车轮设计的优点是简洁性并且非常容易操纵。缺点是稳定性欠佳，这是由于只有 3 个支撑点，因此很容易倾斜。因此，我们在车前面的另一边又安装了一个滚轮。如果车身向前倾斜，另外一个滑轮将着地。

电机、变速箱和车轮

AGV 中选择小型直流电机来驱动。步进电机是另外一个选择，但是它功耗较大。一般地，直流电机的旋转速度很高，因此不适用于 AGV。因此，AGV 中增加了一个变速箱，如图 A4-2 所示。

当电机以稳定的速度 ω_m rad/s 旋转，并且变速箱的变速率为 N，车轮半径为 r 时，AGV 将获得一个稳定的前行速度，计算如下：

$$v=(\omega_m/N)r \text{ m/s}$$

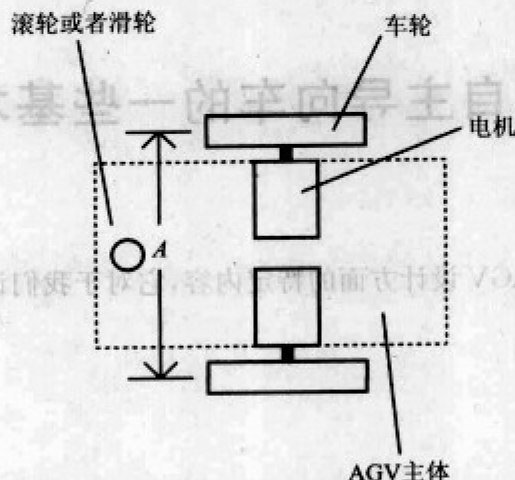


图 A4-1 Derbot 车轮设计

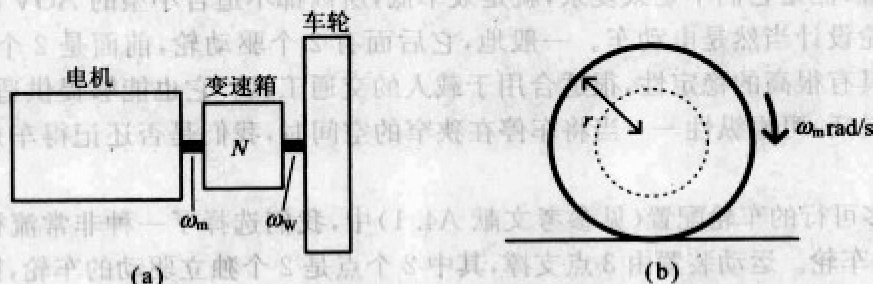


图 A4-2 电机、变速箱和车轮之间的相互作用

转弯的几何计算

运动装置如 AGV 在不同的车轮速度下进行转弯的示意图如图 A4-3 所示。转弯时，2 个车轮分别被驱动并且行驶不同的距离：左车轮为 d_L、右车轮为 d_R。这将导致车身以点 O 为圆心，以 R 为半径转过一个圆弧。车身转过的角度为 θ。由于在驱动车轮时，d_L 和 d_R 是可控的变量，所以它们可以由其他变量推导获得。

由于 d_L 和 d_R 对应的弧度是相同的，我们可以得到下面的等式：

$$\frac{d_L}{(R+A/2)}=\frac{d_R}{(R-A/2)}=\theta$$

推导出：

$$d_L\times(R-A/2)=d_R\times(R+A/2)$$

最后得到：

$$R=\frac{A}{2}\times\frac{(d_L+d_R)}{d_L-d_R}$$

进一步，知道：

$$\theta=\frac{d_L}{(R+A/2)}$$

我们将 R 替换进来，得到：

$$\theta=\frac{(d_L-d_R)}{A}$$

同样，通过三角形 OTS，可以得到车身从最初的参考坐标向前行驶的距离 x_f 为 $R\sin\theta$。

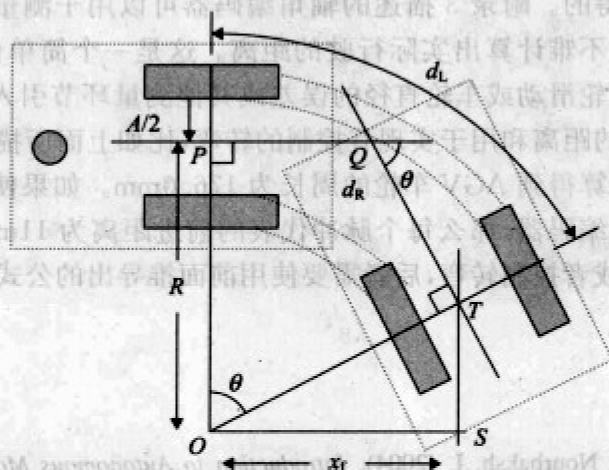
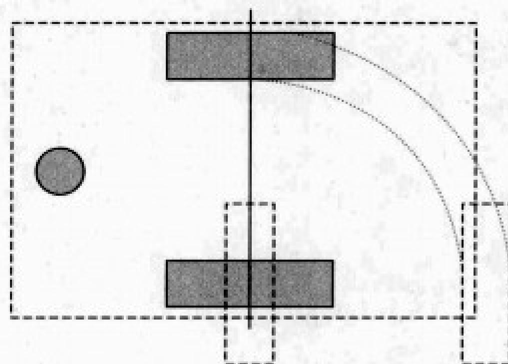


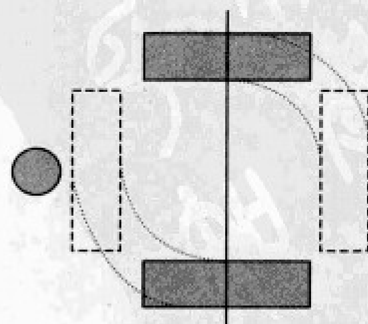
图 A4-3　AGV 转过 $\theta°$（图中没有显示出滑轮的最终位置）

有 2 种重要而特殊的转弯方式，如图 A4-4。一种方式是只有一个车轮被驱动，AGV 将近似地以另外一个车轮为中心来旋转。当转过 90°时，d_L 为 $\pi A/2$，d_R 为 0。这种近似是由于关于旋转点 AGV 的质量不对称，并且滑轮的拖曳也会引起车身滑动。图中显示的只是 90°转弯，但是可以很容易地实现任何角度的转弯。

图 A4-4b 显示的是另外一种转弯方式，其中 2 个车轮各自以相反的方向旋转。这种方式同样会引起滑动。理论上，这种方式转弯时，AGV 是关于 2 个车轮的中间点来旋转的并且 $d_L=-d_R$。



(a) 以一个车轮为支点



(b) 原地转弯——车轮对等旋转

图 A4-4　AGV 转过 90°

在当前的 AGV 实现中，距离 A 的测量值为 16cm。当原地转过 180°时，$R=0$，$d_L=-d_R=\pi A/2=25.1$cm（对于 AGV 来说）。

里程计算

里程计算是用于测量运动装置行驶距离的技术，一般它是通过测量车轮旋转或者其他车轮部件来获得的。附录 3 描述的轴角编码器可以用于测量车轮的角位移。如果车轮直径已知，就不难计算出实际行驶的距离。这是一个简单但相当准确的技术。但是，它没有考虑车轮滑动或车轮直径的误差或其他测量环节引入的误差。里程计算可以用于测量前进的距离和用于实现有控制的转弯，比如上面所描述的。

在附录 3 中，计算得到 AGV 车轮的周长为 176.0mm。如果使用一个“手工制作”的 16 个周期的轴角编码器，那么每个脉冲代表的前进距离为 11mm。利用这些数据，可以实现前进位移或者控制转弯，后者需要使用前面推导出的公式。

参考文献

A4.1. Siegwart, R. and Nourbakhsh, I. (2004). *Introduction to Autonomous Mobile Robots*. MIT Press, Cambridge, MA. ISBN 0-262-19502-X.

540

附录 5 PIC®18 系列指令集(非扩展)

表 A5-1 PIC 18 系列指令集摘要

助记符,操作数		描述	周期	16 位操作码 MSB			LSB	受影响的状态	注
针对字节的文件寄存器操作									
ADDWF	f,d,a	将 WREG 和 f 相加	1	0010	01da0	ffff	ffff	C,DC,Z,OV,N	1,2
ADDWFC	f,d,a	WREG 与 f 及进位相加	1	0010	0da	ffff	ffff	C,DC,Z,OV,N	1,2
ANDWF	f,d,a	WREG 和 f 相与	1	0001	01da	ffff	ffff	Z,N	1,2
CLRF	f,a	f 清零	1	0110	101a	ffff	ffff	Z	2
COMF	f,d,a	f 求补	1	0001	11da	ffff	ffff	Z,N	1,2
CPFSEQ	f,a	比较 f 和 WREG,相等则跳过	1(2 或 3)	0110	001a	ffff	ffff	无	4
CPFSGT	f,a	比较 f 和 WREG,大于则跳过	1(2 或 3)	0110	010a	ffff	ffff	无	4
CPFSLT	f,a	比较 f 和 WREG,小于则跳过	1(2 或 3)	0110	000a	ffff	ffff	无	1,2
DECF	f,d,a	f 减 1	1	0000	01da	ffff	ffff	C,DC,Z,OV,N	1,2,3,4
DECFSZ	f,d,a	f 减 1,为 0 则跳过	1(2 或 3)	0010	11da	ffff	ffff	无	1,2,3,4
DCFSNZ	f,d,a	f 减 1,不为 0 则跳过	1(2 或 3)	0100	11da	ffff	ffff	无	1,2
INCF	f,d,a	f 加 1	1	0010	10da	ffff	ffff	C,DC,Z,OV,N	1,2,3,4
INCFSZ	f,d,a	f 加 1,为 0 则跳过	1(2 或 3)	0011	11da	ffff	ffff	无	4
INFSNZ	f,d,a	f 加 1,不为 0 则跳过	1(2 或 3)	0100	10da	ffff	ffff	无	1,2
IORWF	f,d,a	WREG 和 f 同或	1	0001	00da	ffff	ffff	Z,N	1,2
MOVF	f,d,a	移动 f	1	0101	00da	ffff	ffff	Z,N	1
MOVFF	fs,fd	将 fs(源)送入第一个字 将 fd(目的)送入第二个字	2	1100 1111	ffff ffff	ffff ffff	ffff ffff	无	
MOVWF	f,a	将 WREG 送至 f	1	0110	111a	ffff	ffff	无	
MULWF	f,a	WREG 与 f 相乘	1	0000	001a	ffff	ffff	无	
NEGF	f,a	f 取反	1	0110	110a	ffff	ffff	C,DC,Z,OV,N	1,2
RLCF	f,d,a	f 带进位的循环左移	1	0011	01da	ffff	ffff	C,Z,N	
RLNCF	f,d,a	f 循环左移(不带进位)	1	0100	01da	ffff	ffff	Z,N	1,2
RRCF	f,d,a	f 带进位的循环右移	1	0011	00da	ffff	ffff	C,Z,N	
RRNCF	f,d,a	f 循环右移(不带进位)	1	0100	00da	ffff	ffff	Z,N	
SETF	f,a	置 f	1	0110	100a	ffff	ffff	无	
SUBFWB	f,d,a	f 减去 WREG,带借位	1	0101	01da	ffff	ffff	C,DC,Z,OV,N	1,2

(续)

助记符,操作数		描述	周期	16位操作码 MSB			LSB	受影响的状态	注
针对字节的文件寄存器操作									
SUBWF	f,d,a	f减去WREG	1	0101	11da	ffff	ffff	C,DC,Z,OV,N	
SUBWFB	f,d,a	WREG减去f,带借位	1	0101	10da	ffff	ffff	C,DC,Z,OV,N	1,2
SWAPF	f,d,a	f半字节交换	1	0011	10da	ffff	ffff	无	4
TSTFSZ	f,a	测试f,为0则跳过	1(2或3)	0110	011a	ffff	ffff	无	1,2
XORWF	f,d,a	WREG和f异或	1	0001	10da	ffff	ffff	Z,N	
针对位的文件寄存器操作									
BCF	f,b,a	f的位b清零	1	1001	bbba	ffff	ffff	无	1,2
BSF	f,b,a	f的位b置位	1	1000	bbba	ffff	ffff	无	1,2
BTFSC	f,b,a	检测f的位b,为0则跳过	1(2或3)	1011	bbba	ffff	ffff	无	3,4
BTFSS	f,b,a	检测f的位b,为1则跳过	1(2或3)	1010	bbba	ffff	ffff	无	3,4
BTG	f,d,a	f位翻转	1	0111	bbba	ffff	ffff	无	1,2
控制操作									
BC	n	有进位则跳转	1(2)	1110	0010	nnnn	nnnn	无	
BN	n	为负则跳转	1(2)	1110	0110	nnnn	nnnn	无	
BNC	n	无进位则跳转	1(2)	1110	0011	nnnn	nnnn	无	
BNN	n	非负则跳转	1(2)	1110	0111	nnnn	nnnn	无	
BNOV	n	没有溢出则跳转	1(2)	1110	0101	nnnn	nnnn	无	
BNZ	n	不为0则跳转	2	1110	0001	nnnn	nnnn	无	
BOV	n	溢出则跳转	1(2)	1110	0100	nnnn	nnnn	无	
BRA	n	无条件跳转	1(2)	1101	0nnn	nnnn	nnnn	无	
BZ	n	为0则跳转	1(2)	1110	0000	nnnn	nnnn	无	
CALL	n,s	调用子程序	2	1110	110s	kkkk	kkkk		
		第1个字第2个字		1111	kkkk	kkkk	kkkk	无	
CLRWDT	—	看门狗定时器清零	1	0000	0000	0000	0100	$\overline{TO}$,$\overline{PD}$	
DAW	—	十进制调整WREG	1	0000	0000	0000	0111	C	
GOTO	n	跳转到地址	2	1110	1111	kkkk	kkkk	无	
		第1个字第2个字		1111	kkkk	kkkk	kkkk		
NOP	—	无操作	1	0000	0000	0000	0000	无	
NOP	—	无操作	1	1111	xxxx	xxxx	xxxx	无	
POP	—	弹出返回栈顶层(TOS)	1	0000	0000	0000	0110	无	
PUSH	—	压入返回栈顶层(TOS)	1	0000	0000	0000	0101	无	
RCALL	n	相对调用	2	1101	1nnn	nnnn	nnnn	无	
RESET		软件设备复位	1	0000	0000	1111	1111	所有	
RETFIE	s	中断使能返回	2	0000	0000	0001	000s	GIE/GIEH,PEIE/GIEL	
RETLW	k	立即数送到WREG,子程序返回	2	0000	1100	kkkk	kkkk	无	

(续)

助记符,操作数		描述	周期	16 位操作码				受影响的状态	注
				MSB			LSB		
控制操作									
RETURN	s	子程序返回	2	0000	0000	0001	001s	无	
SLEEP	—	进入休眠模式	1	0000	0000	0000	0011	$\overline{TO}$,$\overline{PD}$	
立即数操作									
ADDLW	k	立即数和 WREG 相加	1	0000	1111	kkkk	kkkk	C,DC,Z,OV,N	
ANDLW	k	立即数和 WREG 相与	1	0000	1011	kkkk	kkkk	Z,N	
IORLW	k	立即数与 WREG 同或	1	0000	1001	kkkk	kkkk	Z,N	
LFSR	f,k	将立即数(位 12)第 2 个字送入 FSRx 第 2 个字	2	1110 1111	1110 0000	00ff kkkk	kkkk kkkk	无	
MOVLB	k	立即数送入 BSR<3:0>	1	0000	0001	0000	kkkk	无	
MOVLW	k	立即数送到 WREG	1	0000	1110	kkkk	kkkk	无	
MULLW	k	将立即数和 WREG 相乘	1	0000	1101	kkkk	kkkk	无	
RETLW	k	立即数送到 WREG,子程序返回	2	0000	1100	kkkk	kkkk	无	
SUBLW	k	立即数减去 WREG	1	0000	1000	kkkk	kkkk	C,DC,Z,OV,N	
XORLW	k	立即数和 WREG 异或	1	0000	1010	kkkk	kkkk	Z,N	
数据存储器←→程序存储器操作									
TBLRD*		读表	2	0000	0000	0000	1000	无	
TBLRD*+		读表后再递增		0000	0000	0000	1001	无	
TBLRD*-		读表后再递减		0000	0000	0000	1010	无	
TBLRD+*		先递增再读表		0000	0000	0000	1011	无	
TBLWT*		写表	2(5)	0000	0000	0000	1100	无	
TBLWT*+		写表后再递增		0000	0000	0000	1101	无	
TBLWT*-		写表后再递减		0000	0000	0000	1110	无	
TBLWT+*		先递增再写表		0000	0000	0000	1111	无	

注:1. 当 I/O 寄存器用自身内容修改自身时(例如:MOVF PORTB,1),使用的值将是出现在引脚上的值,而非锁存器中的值。例如,如果一引脚配置为输入,其数据锁存器中的值为“1”,但此时外部器件将该引脚拉为低电平,则被写回数据锁存器的数据值将是“0”。

2. 如果预分频器被分配给 Timer 0 模块,那么对 TMR0 寄存器执行这条指令(适当时,d=1)将清零预分频器。

3. 如果程序计数器(PC)被修改或者执行一个条件检测为真的跳转,则指令需要 2 个指令周期。第 2 个指令周期执行 1 条空操作 NOP 指令。

4. 一些指令是 2 字指令的。这些指令的第 2 个字作为一个 NOP 指令来执行,除非指令的第 1 个字需要获取嵌在第 2 个字中的信息。这保证了所有的存储器单元都有 1 个有效指令。

5. 如果写表指令启动写内部存储器的周期,写操作将继续下去直到结束。

表 A5-2　操作码的操作数符号总结

符号	描述(下划线的值为MPLAB汇编器默认值)
a	RAM访问位 a=0:存储单元在可读取RAM中 <u>a=1</u>:RAM区由区选择位(BSR)指定
b	字节中的位号
d	目的选择位 d=0:结果存储在W寄存器中 <u>d=1</u>:结果存储在文件寄存器f中(即数据存储器)
f	8位的数据存储器地址
fd	12位的数据存储器地址,数据移动的目的地址
fs	12位的数据存储器地址,数据移动的源地址
k	立即数,常数或者标号(8位、12位或者20位)
n	相对跳转指令的相对地址(二进制补码),或者Call和Return指令的直接地址
s	快速Call/Return模式选择位 <u>s=0</u>:不会更新到阴影寄存器或者从其中更新 s=1:更新W、Status和BSR寄存器到隐藏寄存器或者从其中更新

附录 6　C 语言要点

本附录以一组表格的形式简要地提供了 C 语言编程方面的一些关键信息。大部分 C 语言特性的示例和进一步解释可以在本书第 14 章～第 19 章找到。

表 A6-1　数据类型和结构体定义相关的 C 语言关键字

关键字	含义摘要	关键字	含义摘要
char	单个字符，通常为 8 位	signed	应用到 char 或者 int 的修饰符（char 和 int 默认是有符号的）
const	不能修改的数据	sizeof	返回指定项的字节个数，指定项可以是变量、表达式或者数组
double	"双精度"浮点数	struct	定义一个数据结构
enum	定义一个只能表示特定整数的变量	typedef	重新命名已存在的数据类型
float	"单精度"浮点数	union	共享 2 个或多个任意数据类型变量的存储块
int	整型数	unsigned	应用到 char 或者 int 的修饰符（char 和 int 默认是有符号的）
long	扩展的整型数；如果单独使用，默认为整型	void	没有值或类型
short	短整型数；如果单独使用，默认为整型	volatile	变量可被程序代码之外的因素改变

表 A6-2　程序流相关的 C 语言关键字

关键字	含义摘要	关键字	含义摘要
break	引发程序退出循环	for	定义一个重复循环——只要 for 的条件保持为真，循环将执行
case	在 switch 表达式中指定用于选择的项	goto	程序执行到标号所在的语句
continue	跳过后续程序直接运行到 for、while 或者 do 语句的结尾处	if	开始一个条件语句；如果条件为真，将执行相应的语句
default	指定 switch 表达式的默认选择项，用于没有选项匹配时	return	返回到调用程序处，也会返回由函数指定的任意值
do	与 while 一起创建一个循环。当 while 表达式为真时，do 后面的语句将重复执行	switch	同 case 一起使用，用于对多个选择项进行选择；switch 语句的表达式同多个 case 选项进行比较
else	与 if 一起使用。如果 if 条件为假，else 语句将执行	while	定义一个重复循环——只要 while 的条件保持为真，将执行循环

表 A6-3 数据存储类型相关的 C 语言关键字

关键字	含义摘要	关键字	含义摘要
auto	变量只存在于它声明的块中。它是默认类型	register	变量存放在 CPU 的一个寄存器中;因此,取地址操作符(&)不起作用
extern	声明一个定义在其他地方的变量	static	声明一个在整个程序执行期间都存在的变量;变量的作用范围是程序中可以引用到它的地方

表 A6-4 MPLAB C18 C 编译器实现的 C 语言数据类型

数据类型	描述	长度(字节)	范围
char	字符	1	−128～+127
signed char	字符	1	−128～+127
unsigned char	字符	1	0～255
int	整数	2	−32 768～+32 767
unsigned int	整数	2	0～65 535
short	整数	2	−32 768～+32 767
unsigned short	整数	2	0～65 535
short long	整数	3	−8 388 608～8 388 607
unsigned short long	整数	3	0～16 777 215
long	整数	4	−2 147 483 648～2 147 483 647
unsigned long	整数	4	0～4 294 967 295
float	浮点	4	1.17549×10^{-38}～$6.80565\times10^{+38}$
double	浮点,双精度	4	1.17549×10^{-38}至$6.80565\times10^{+38}$

表 A6-5 C 操作符

优先级和顺序	操作	符号	优先级和顺序	操作	符号
括号和数据访问					
1,从左至右	函数调用	()	1,从左至右	指向成员	X−>Y
1,从左至右	数组下标	[]	1,从左至右	选择成员	X.Y
算术操作符					
4,从左至右	加法	X+Y	3,从左至右	乘法	X*Y
4,从左至右	减法	X−Y	3,从左至右	除法	X/Y
2,从右至左	一元加法	+X	3,从左至右	取模	%
2,从右至左	一元减法	−X			
关系操作符					
6,从左至右	大于	X>Y	6,从左至右	小于等于	X<=Y
6,从左至右	大于等于	X>=Y	7,从左至右	等于	X==Y
6,从左至右	小于	X<Y	7,从左至右	不等于	X!=Y

（续）

优先级和顺序	操作	符号	优先级和顺序	操作	符号
逻辑操作符					
11，从左至右	与（当X和Y都不为0时，为1）	X&&Y	2，从右至左	取反（当X为0时，值为1）	！X
12，从左至右	或（当X或Y不为0时，为1）	X||Y			
位操作符					
8，从左至右	逻辑与	X&Y	2，从左至右	求补（逻辑取反）	~X
10，从左至右	逻辑或	X|Y	5，从左至右	右移。X右移Y次	X≫Y
9，从左至右	逻辑异或	X^Y	5，从左至右	左移。X左移Y次	X≪Y
赋值操作符					
14，从右至左	赋值	X=Y	14，从右至左	逻辑与赋值	X&=Y
14，从右至左	加法赋值	X+=Y	14，从右至左	逻辑同或赋值	X|=Y
14，从右至左	减法赋值	X-=Y	14，从右至左	逻辑异或赋值	X^=Y
14，从右至左	乘法赋值	X*=Y	14，从右至左	右移赋值	X≫=Y
14，从右至左	除法赋值	X/=Y	14，从右至左	左移赋值	X≪=Y
14，从右至左	余数赋值	X%=Y			
递增和递减操作符					
2，从右至左	先递增	++X	2，从右至左	后递增	X++
2，从右至左	先递减	--X	2，从右至左	后递减	X--
条件操作符					
13，从右至左	等于X(Z≠0)或Y(Z=0)	Z? X：Y	15，从左至右	先计算X，再计算Y	X，Y
"数据解析"操作符					
2，从右至左	由X指向的对象或函数	*X	2，从右至左	X的地址	&X
2，从右至左	使用指定的（标量）类型对X的值进行类型转换	（类型）X	2，从右至左	X的大小，字节为单位	X类型

关键字：Prec.=先；L=左；R=右

表 A6-6 预处理机伪指令示例

关键字	含义摘要	关键字	含义摘要
#if	根据表达式的取值来条件编译代码。必须以#endif 结束	#define	定义一个用于源代码中的字符常串，在代码行编译之前被替换
#ifdef	与#if 类似，但是它是测试指定的符号是否已经定义。以#endif 结束	#error	产生用户定义的错误消息
#ifndef	同#ifdef，但是它是测试指定的符号是否没有定义	#include	在当前位置包括指定文件的全部内容，文件可以包含不受限制的 C 语言代码；之后该文件同源程序一起编译
#else	同#ifdef 一起使用，提供另外一段用于编译的代码	#line	返回到调用程序处，也会返回由函数指定的任意值
#elif	在#if 代码中使用，用于测试一个新的表达式	#pragma	用于传递伪指令类似的进一步信息给预处理机，一般是与编译器相关的信息
#endif	结束#if 代码段	#undef	取消#define 定义的字符常串

表 A6-7 标点符号的一些应用

符 号	应 用	示 例
:	结束一个标号	loop:
:	用于位域格式(在表 A6-5 中也可看到)	unsigned RB0:1;
;	结束一个语句或者声明	unsigned RB0:1;
.	指定结构体中的成员(在表 A6-5 中也可看到)	PORTAbits. RA2 = 1;
\	下一行是该行的继续	—
{}	定义一个代码块	—

表 A6-8 MPLAB C18 新增的存储器修饰符

	ROM	RAM
far	变量可以在程序存储器的任何地方	变量可以在数据存储器的任何地方；访问时需要使用区转换指令，far 和 RAM 是默认组合
near	变量在程序存储器中，地址<64KB	变量在可访问 RAM 中

表 A6-9 MPLAB C18 存储模型

存储模型	指针大小	默认的 ROM 修饰符	存储器大小
small(默认)	16 位	near	程序存储器≤64KB
large	24 位	far	程序存储器>64KB

索 引

索引中的页码为英文原书页码，与本书中页边标注的页码一致。

C

D

K

L

T

Designing Embedded Systems with PIC Microcontrollers
Principles and Applications

PIC嵌入式系统开发

“我买了所有PIC相关的书，本书的详尽和透彻令其他同类图书难以望其项背……这毋庸置疑是PIC领域的圣经。”

——Amazon.com读者评论

“本书是电子工程、机电一体化和计算机工程专业学生的理想教材，也是专业技术人员极具价值的参考书。”

——Microchip公司网站

本书是广受赞誉的嵌入式系统著作。以Microchip公司3款PIC系列微控制器为实例，循序渐进、系统全面地阐述了嵌入式系统设计的思想与实践。不仅讨论了各微控制器和外围设备，还涵盖了汇编语言和C语言编程、人机接口、串行和并行通信、数据采集与处理、网络互连以及一个实时操作系统。

书中配有大量的图、表和示例，图文并茂，叙述生动，可读性强。此外，本书分为初中高三个部分，可以根据实际情况灵活选择阅读和教学顺序。通过对本书的学习，读者可以在较短时间内轻松掌握当今嵌入式系统软硬件的必备知识和技能，并能举一反三，融会贯通。

本书原版配套网站 http://www.embedded-knowhow.co.uk/book2.htm 提供勘误、补充信息、教学计划、教学课件和考试样卷。

Tim Wilmshurst 英国德比大学教授，并长期任教于剑桥大学。IET（英国工程技术学会，前身为IEE）会士，著名的嵌入式系统专家。主要研究方向为电子技术和嵌入式系统，在PIC微控制器的应用开发上有很深的造诣。他在本书中作为实例设计的自动导向车——Derbot AGV已经广泛应用于嵌入式系统教学，获得了巨大成功。



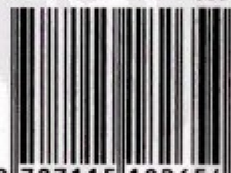
本书译自原版*Designing Embedded Systems with PIC Microcontrollers: Principles and Applications*，并由Elsevier授权出版。



本书相关信息请访问：图灵网站 http://www.turingbook.com
读者/作者热线：(010) 88593802
反馈/投稿/推荐信箱：contact@turingbook.com

分类建议 电子电气/嵌入式开发

人民邮电出版社网址 www.ptpress.com.cn


ISBN 978-7-115-18265-4
9 787115 182654 >
ISBN 978-7-115-18265-4/TN
定价：69.00 元